# Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

## MIPS Assembly

## Lab Schedule

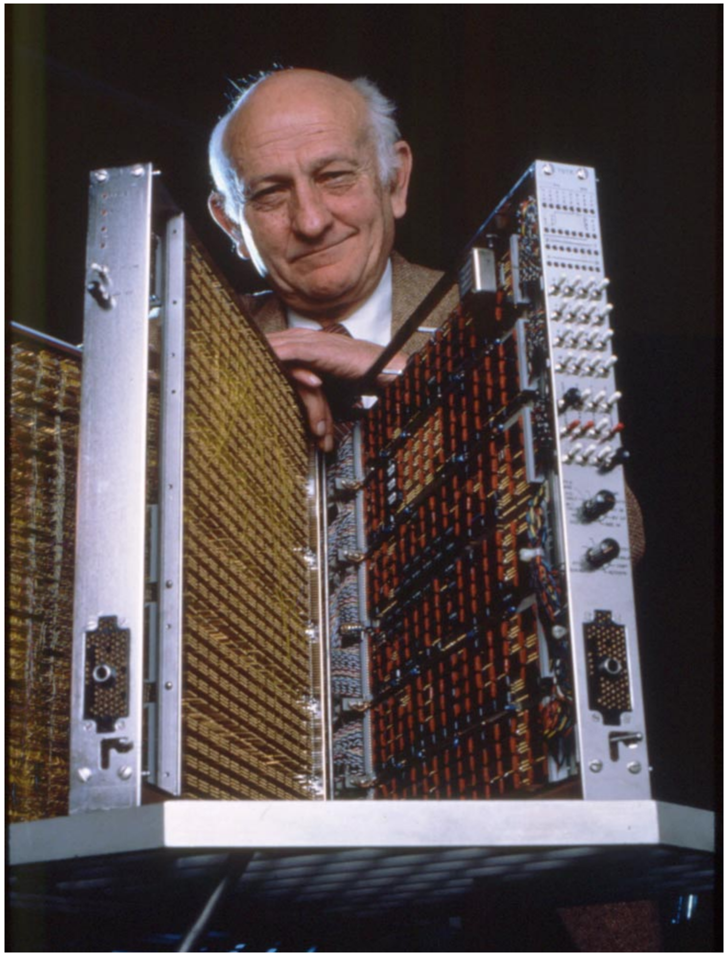
### Activities

- **This Week**
  - MIPS discussion
  - Practice problems (whiteboard)
  - Using the QtSPIM simulator
  - Discuss available resources

### Assignments Due

- **Lab 10**
  - **Due by Apr 12th 5:00am**
- **Lab 11**
  - **Due by Apr 19th 5:00am**
- **Lab 12**
  - **Due by May 3rd 5:00am**

## Person of the Day – John Cocke



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

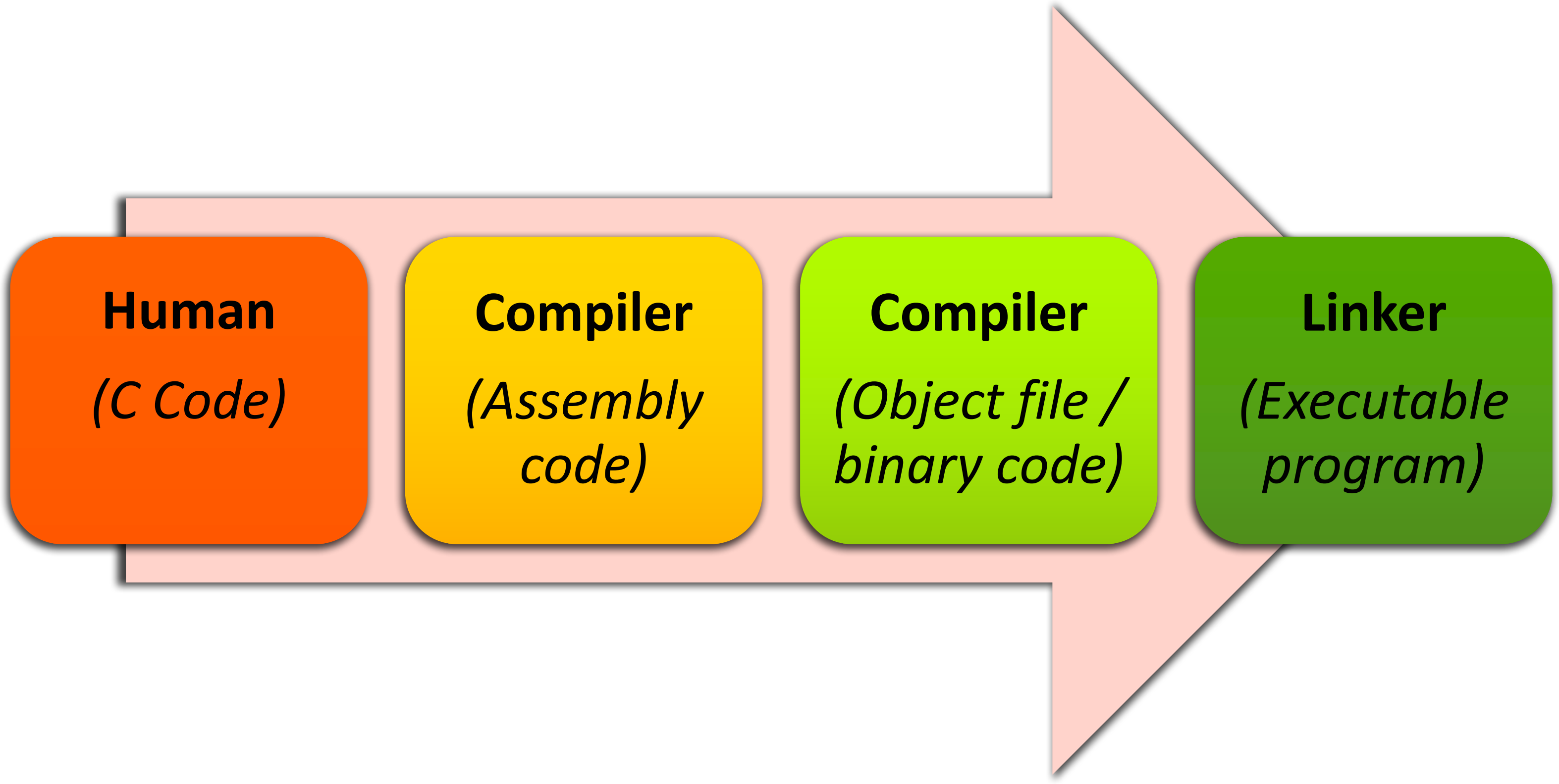
**RISC = Reduced Instruction Set Computing**

Achieve higher performance with simple instructions that execute faster

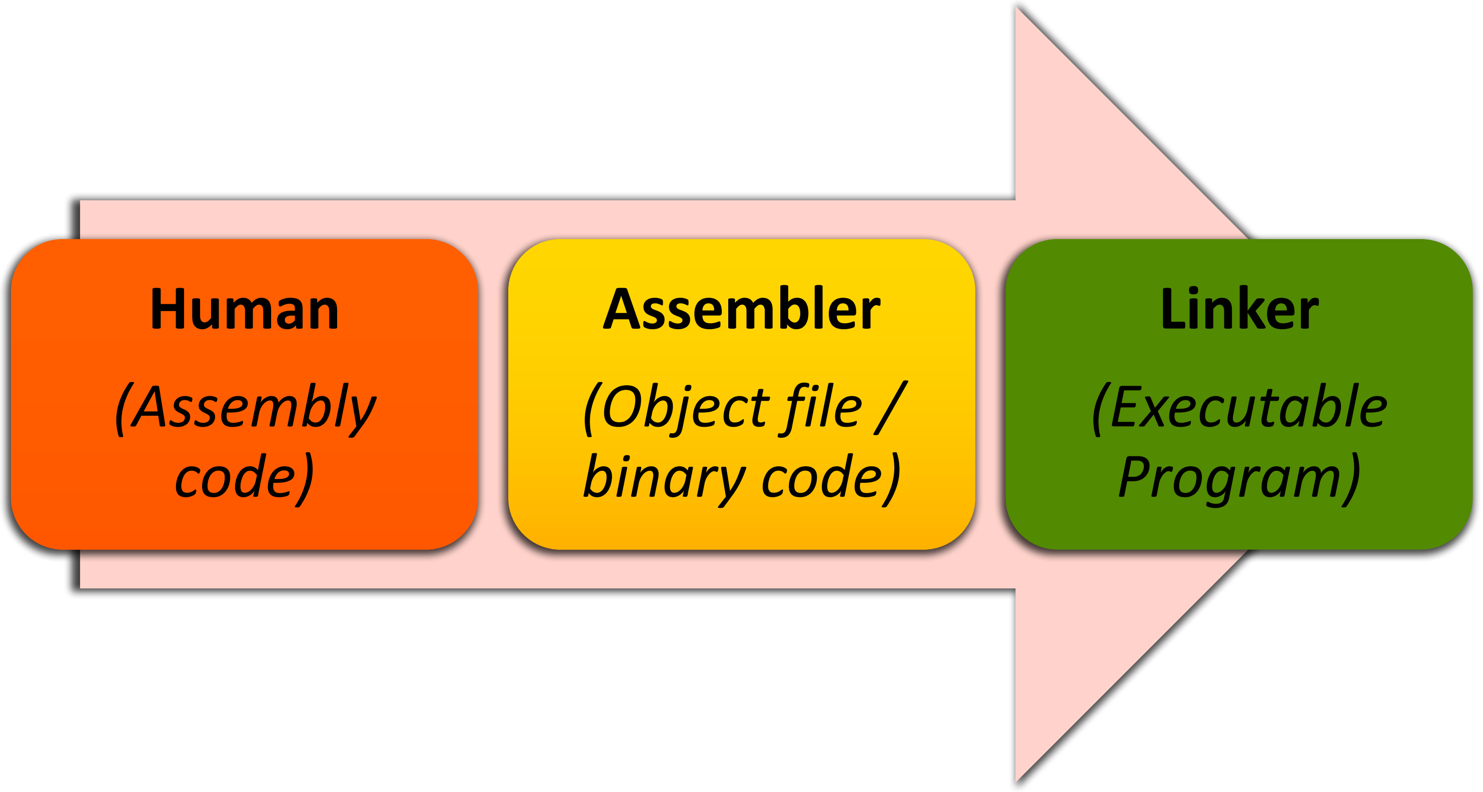
## Person of the Day – John Hennessy

- 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


**Human**
*(Assembly code)*
**Assembler**
*(Object file / binary code)*
**Linker**
*(Executable Program)*

## MIPS

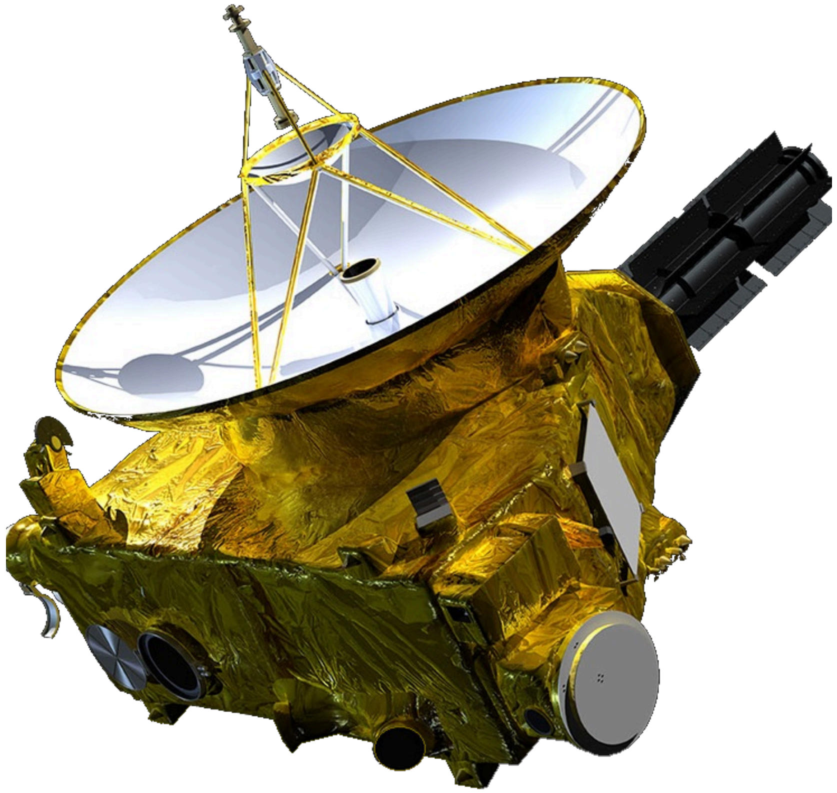
## MIPS Overview

- 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…

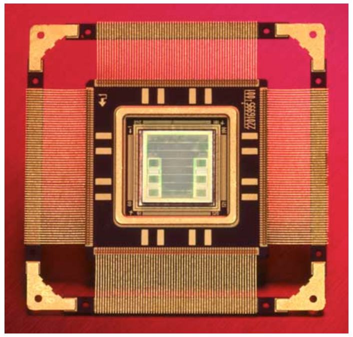
## MIPS Products

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

## MIPS Products



- **NASA New Horizons probe**
  - Launched January 2006
- MIPS "Mongoose-V" chip
  - 12 MhZ *(2006, remember?)*
  - **Radiation Hardened**
  - Based on R3000 (PlayStation CPU)



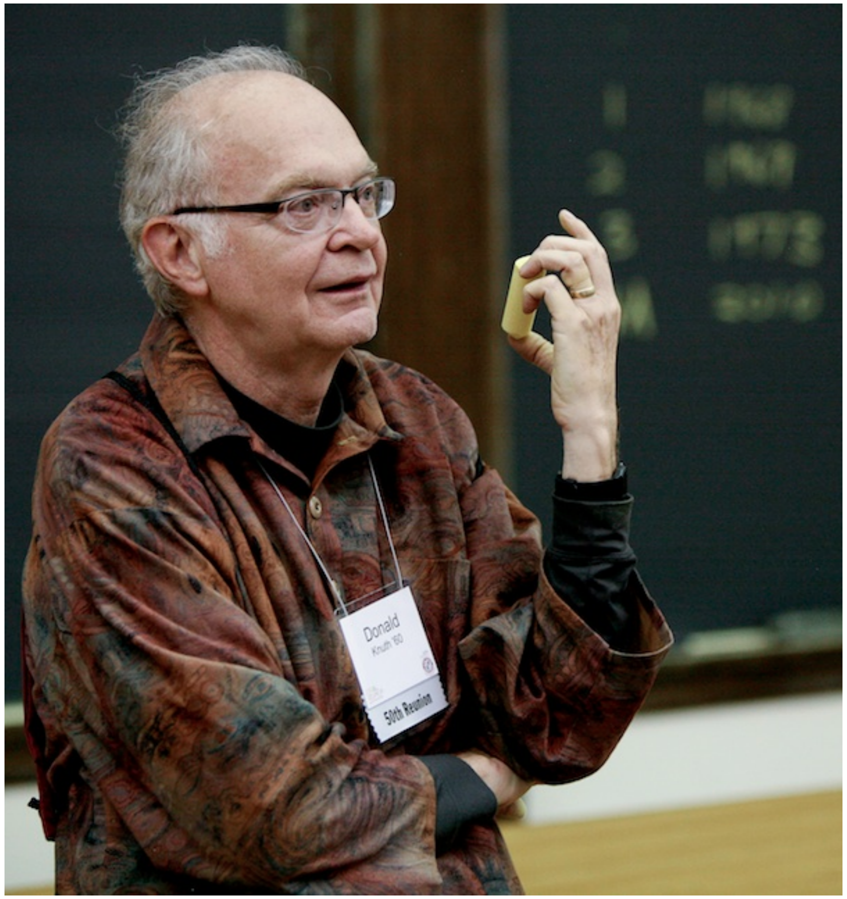
http://blog.imgtec.com/mips-processors/mips-goes-to-pluto
http://synova.com/proc/MongooseV.pdf

## MIPS Design

- RISC – **What does this mean?**
  - **R**educed **I**nstruction **S**et **C**omputing
  - 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

This is your motivation in the assembly labs!

## 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 December 2015 is **3883 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

$$\underset{\uparrow \atop \text{Operation / “Op code”}}{\texttt{sub}}\ \underbrace{\texttt{<result>, <input1>, <input2>}}_{\text{Operands}}$$

## Task : Write Code

- **Write MIPS assembly for**

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

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

**<u>Congratulations!</u>**

You're now an assembly programming expert!

## Data Sources

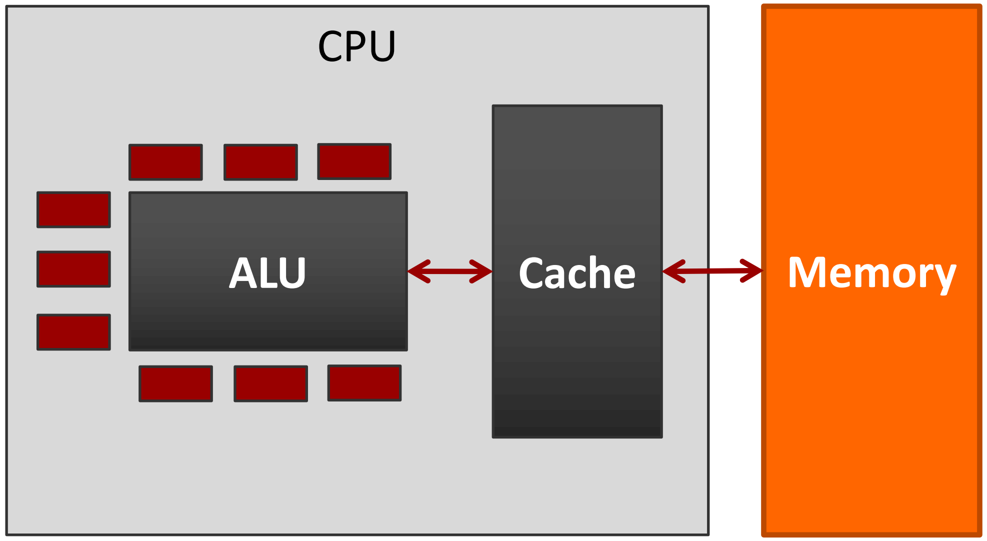
- Previous example was *(just a little bit)* fake…
  - We made up some variables: `temp0`, `temp1`, `f`, `g`, `h`, `i`, and `j`
  - This is what you do when programming in C++ (or any high level language)

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

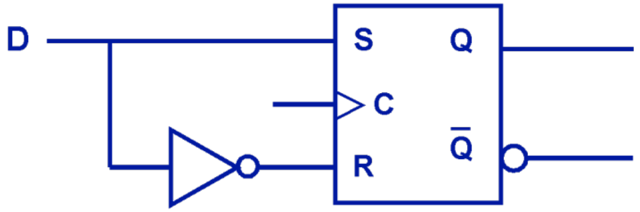
## Aside – Cache

- **Review: Does the programmer explicitly manage the cache?**
- **Answer: 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

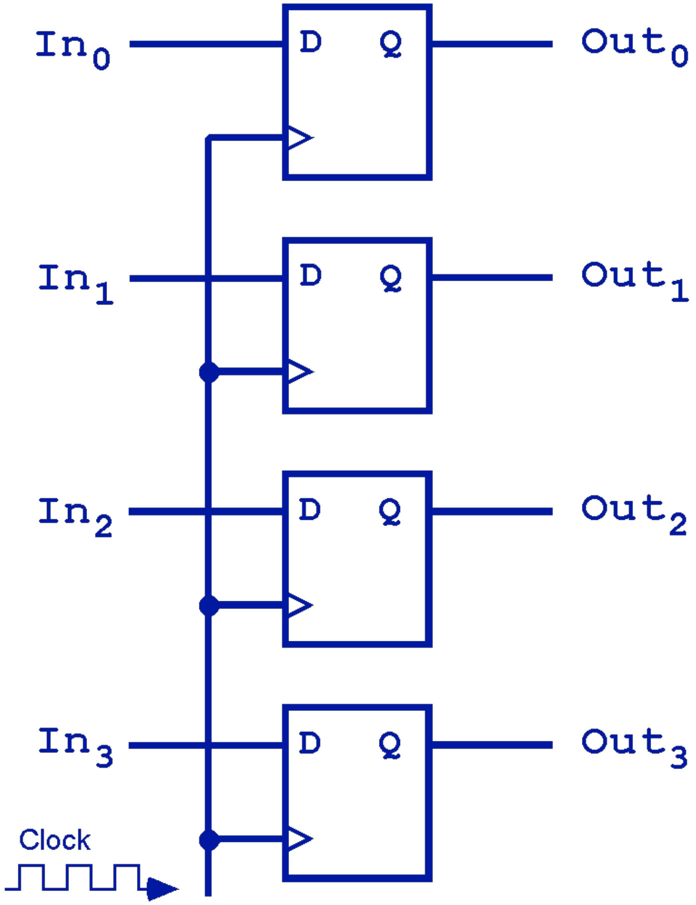
## ECPE 71

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

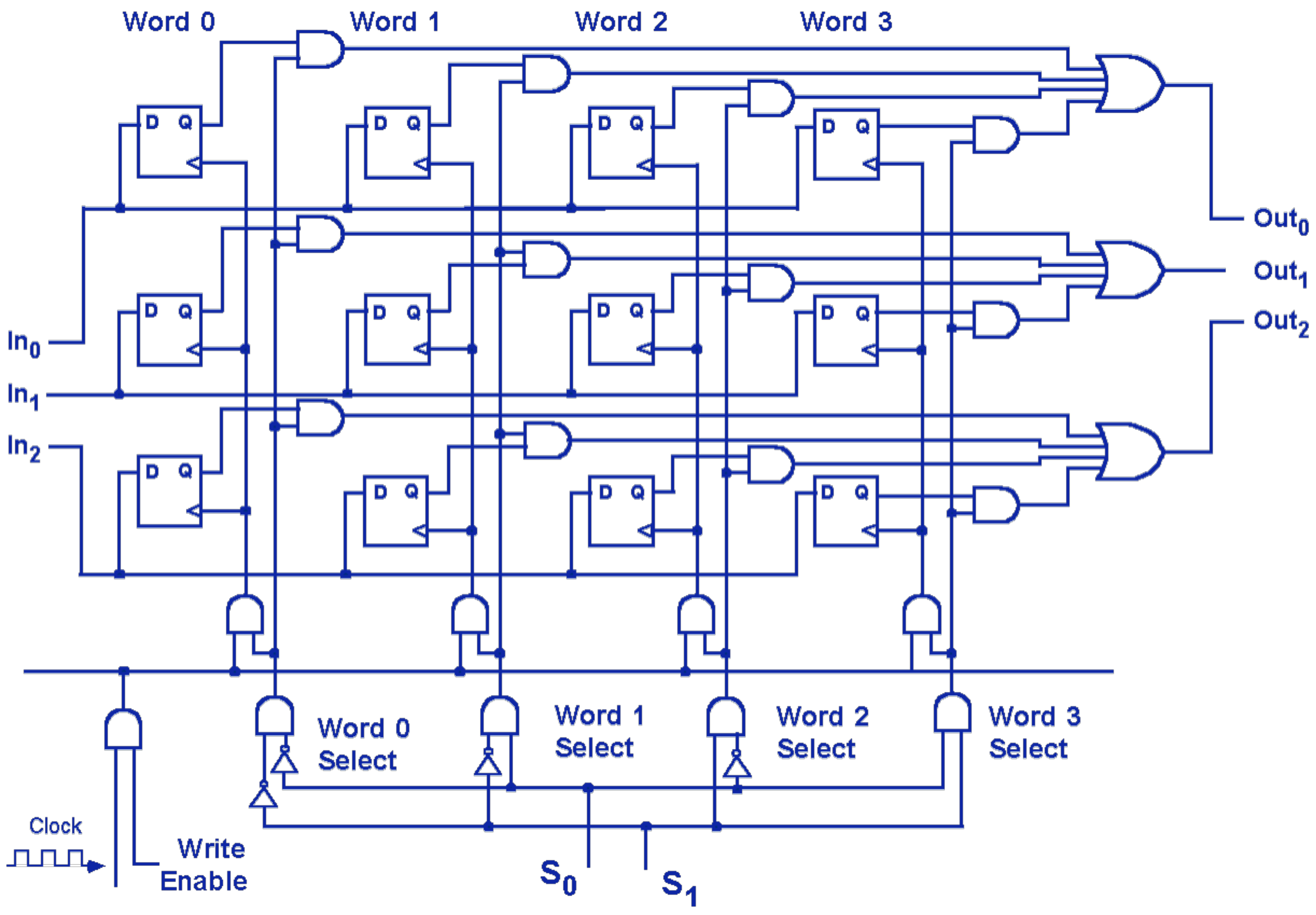
Flip Flops! *(D Flip Flop shown)*



| 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 <u>using registers</u> 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)

This is a common design in *RISC-style* machines: a *load-store* architecture

## 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**!

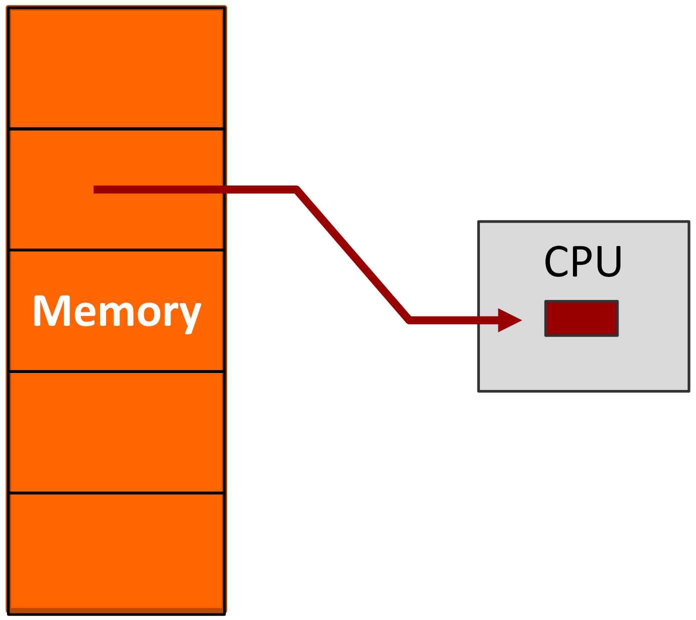
## Memory Fundamentals

- 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

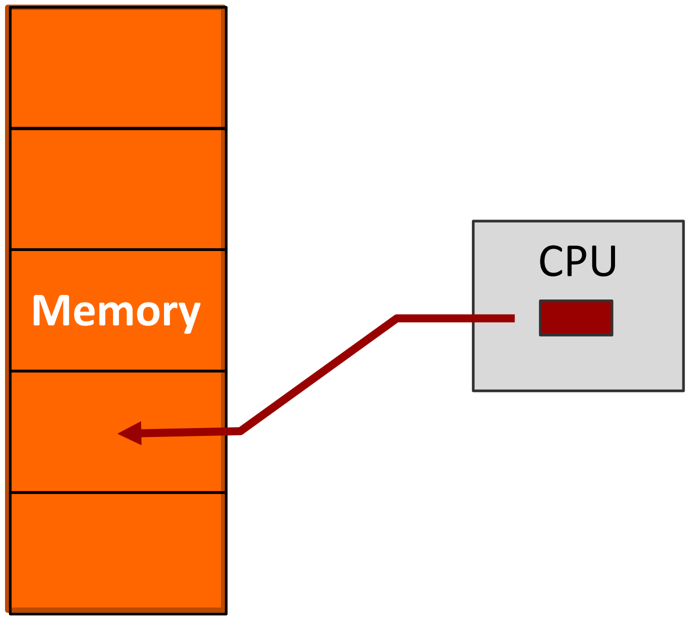
**Load**

- Copy data from memory to register



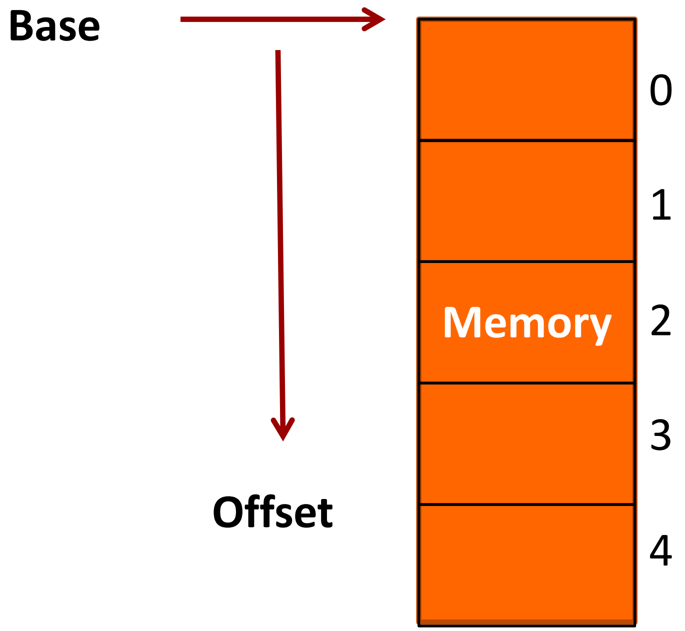
**Store**

- 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!
  - Will need a loop that loads 1 element at a time…

## 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>)
```

$\texttt{sb}\ \underset{\uparrow\ \text{Register}}{\texttt{<reg>}},\ \underbrace{\texttt{<offset>(<base addr reg>)}}_{\text{Memory Location}}$

## 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 + array[16]

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

**Map:**

$s1 = g
$s2 = h
$s3 = 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 16th 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 + 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?**
- **Answer: It depends**
- **Compiler will choose** where to place variables
  - Registers: Loop counters, frequently accessed scalar values, variables local to a procedure
  - Memory: Arrays, infrequently accessed data values

## MIPS Branches / Loops

## Branches, Tests, Jump

- Branch on Equal (if $1 == $2, goto dest)

```
beq <reg1>, <reg2>, <destination>
```

- Set on Less Than (if $2 < $3, set $1 = 1, otherwise 0)

```
slt <reg1>, <reg2>, <reg3>
```

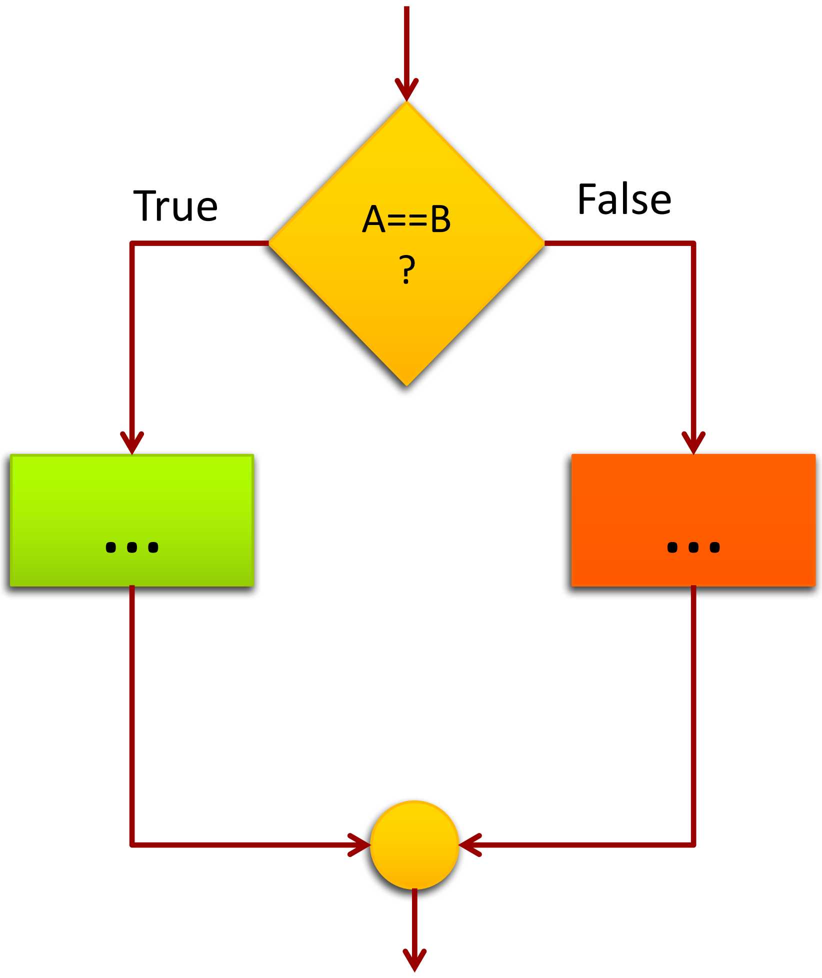
- Jump (goto dest)

```
j <destination>
```

## Task : Write Code

- **Write MIPS assembly for:**

```
if (A == B)
{
      <equal-code>
}
else
{
      <not-equal-code>
}
<after-if-code>
```



## Task : Write Code

- **Write MIPS assembly:**

**Map:**
$s0 = A
$s1 = B

**Code:**

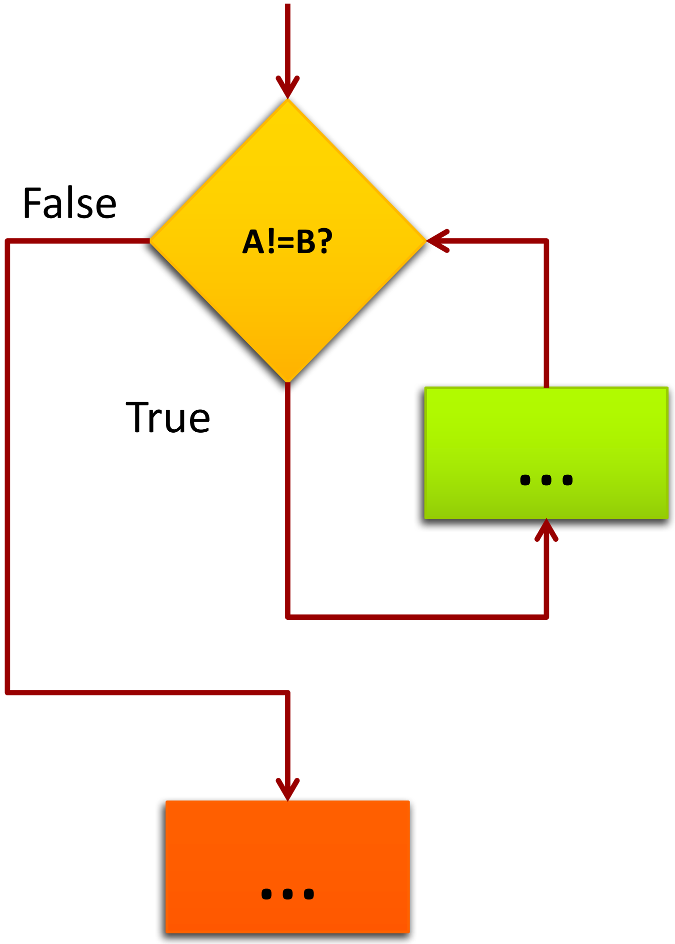
```
        beq $s0,$s1,equal
        <not-equal-code>
        j done
equal:  <equal-code>
        j done
done:   <after-if-code>
```

## Task : Write Code

- **Write MIPS assembly for:**

```
while (A != B)
{
      <loop-body>
}

<post-loop-code>
```



## Task : Write Code

- **Write MIPS assembly:**

**Map:**
$s0 = A
$s1 = B

**Code:**

```
start:   beq $s0,$s1,done
         <loop-body>
         j start
done:    <post-loop-code>
```

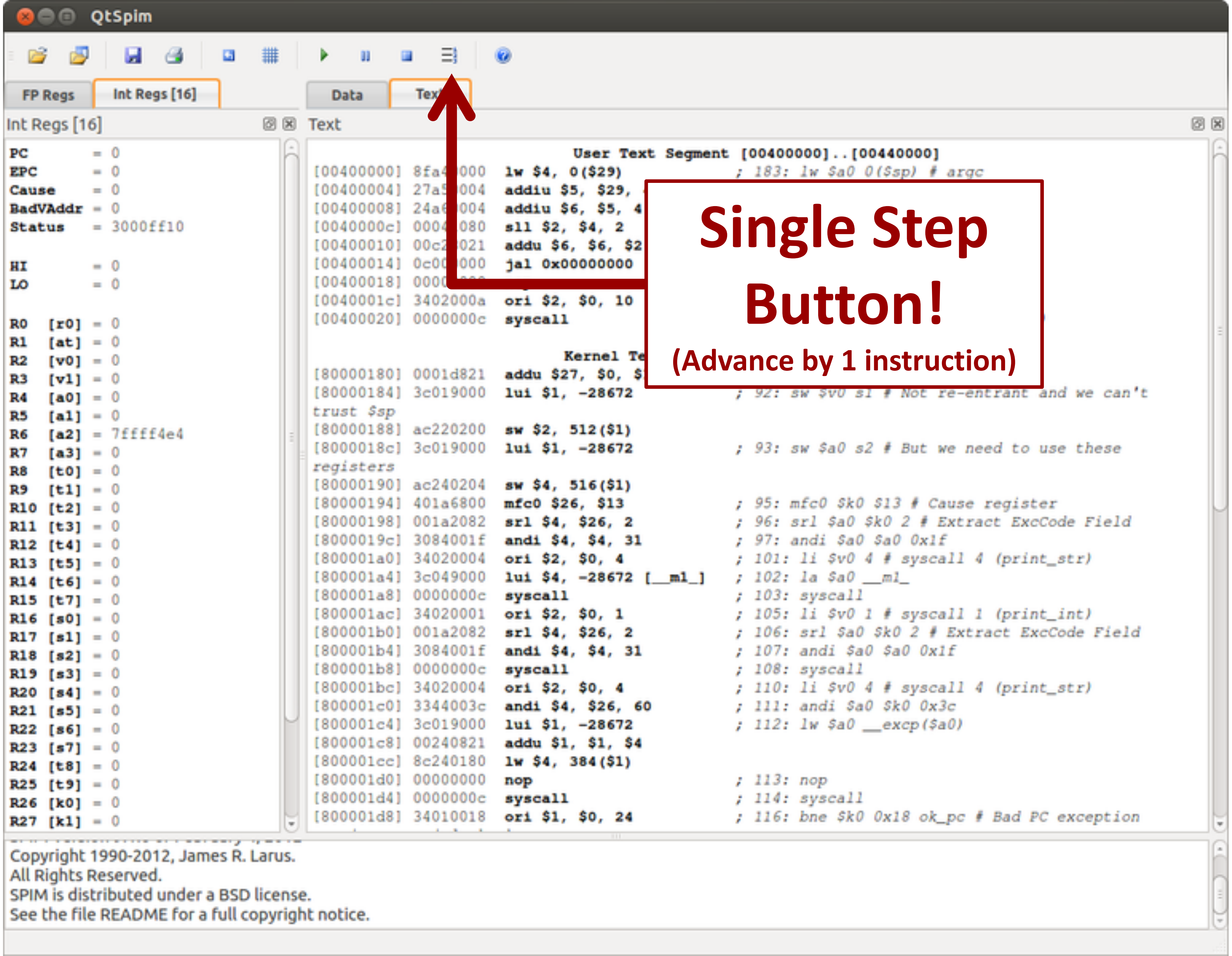
There are many, **many**, variations of branch or test instructions intended to simplify programming

1. Show: Appendix A Reference
2. Discuss: Instruction versus *Pseudo-Instruction*

## 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, …

## MIPS Simulator Walkthrough



**Single Step Button!**

**(Advance by 1 instruction)**

```
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 0
R5  [a1] = 0
R6  [a2] = 7ffff4e4
R7  [a3] = 0
R8  [t0] = 0
R9  [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
```

```
                    User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)           ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29,
[00400008] 24a60004  addiu $6, $5, 4
[0040000c] 00041080  sll $2, $4, 2
[00400010] 00c23021  addu $6, $6, $2
[00400014] 0c000000  jal 0x00000000
[00400018] 0000
[0040001c] 3402000a  ori $2, $0, 10
[00400020] 0000000c  syscall

                    Kernel Te
[80000180] 0001d821  addu $27, $0, $
[80000184] 3c019000  lui $1, -28672          ; 92: sw $v0 s1 # Not re-entrant and we can't
trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672          ; 93: sw $a0 s2 # But we need to use these
registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13           ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2          ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31         ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4           ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]  ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                 ; 103: syscall
[800001ac] 34020001  ori $2, $0, 1           ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082  srl $4, $26, 2          ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f  andi $4, $4, 31         ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c  syscall                 ; 108: syscall
[800001bc] 34020004  ori $2, $0, 4           ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c  andi $4, $26, 60        ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000  lui $1, -28672          ; 112: lw $a0 __excp($a0)
[800001c8] 00240821  addu $1, $1, $4
[800001cc] 8c240180  lw $4, 384($1)
[800001d0] 00000000  nop                     ; 113: nop
[800001d4] 0000000c  syscall                 ; 114: syscall
[800001d8] 34010018  ori $1, $0, 24          ; 116: bne $k0 0x18 ok_pc # Bad PC exception
```

```
Copyright 1990-2012, James R. Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
```