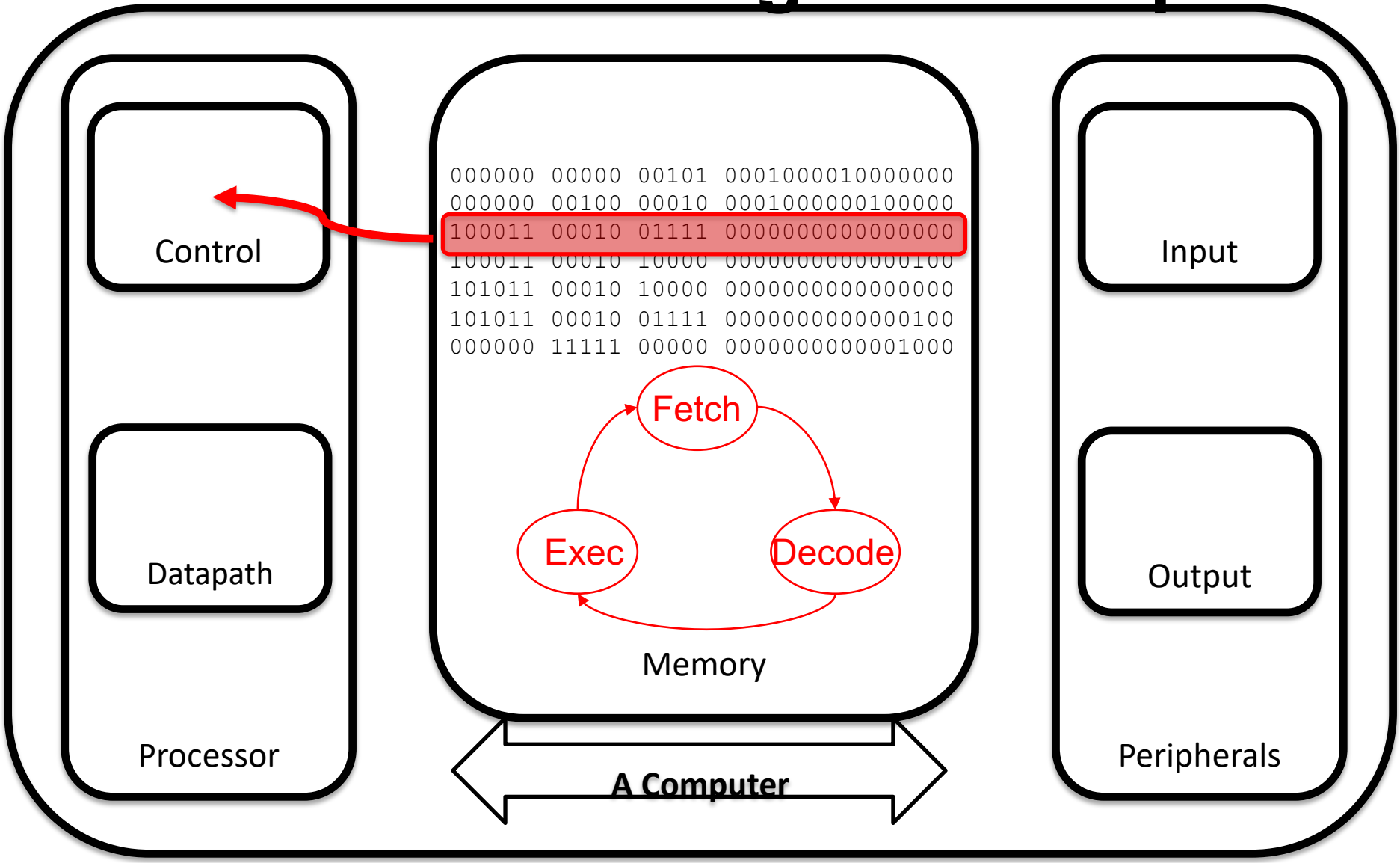
CprE 381: Computer Organization and Assembly Level Programming

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- HW0 due TONIGHT by 11:59pm
  - hard deadline
  - multiple submissions allowed, so submit early and often if you are working up to the deadline
- HW1 posted (due next Mon Jan 28)
- Lab 1
  - Files due in Canvas prior to the start of lab
  - Demo remaining components in lab
- Lab 2
  - Prelab must be completed ***prior*** to start of lab

Review: Stored Program Computer



C Code Example

Simple C procedure: sum_pow2=2^{b+c}

```
 1: int sum_pow2 (int b, int c)
 2: {
 3:   int pow2[8] = {1, 2, 4, 8, 16, 32, 64, 128};
 4:   int a, ret;
 5:   a = b + c;
 6:   if (a < 8)
 7:      ret = pow2[a];
 8:   else
 9:      ret = 0;
10:   return(ret);
11: }
```

Review: Constants

- Often want to be able to specify operand in the instruction: immediate or literal
- Use the `addi` instruction

```
addi dst, src, immediate
```

- The immediate is a 16 bit signed value between **-2^{15} and $2^{15}-1$**
- Sign extended to 32 bits
- Consider the following C code

```
a++;
```

- Implemented using the `addi` operator

```
addi $s0, $s0, 1    # a = a + 1
```

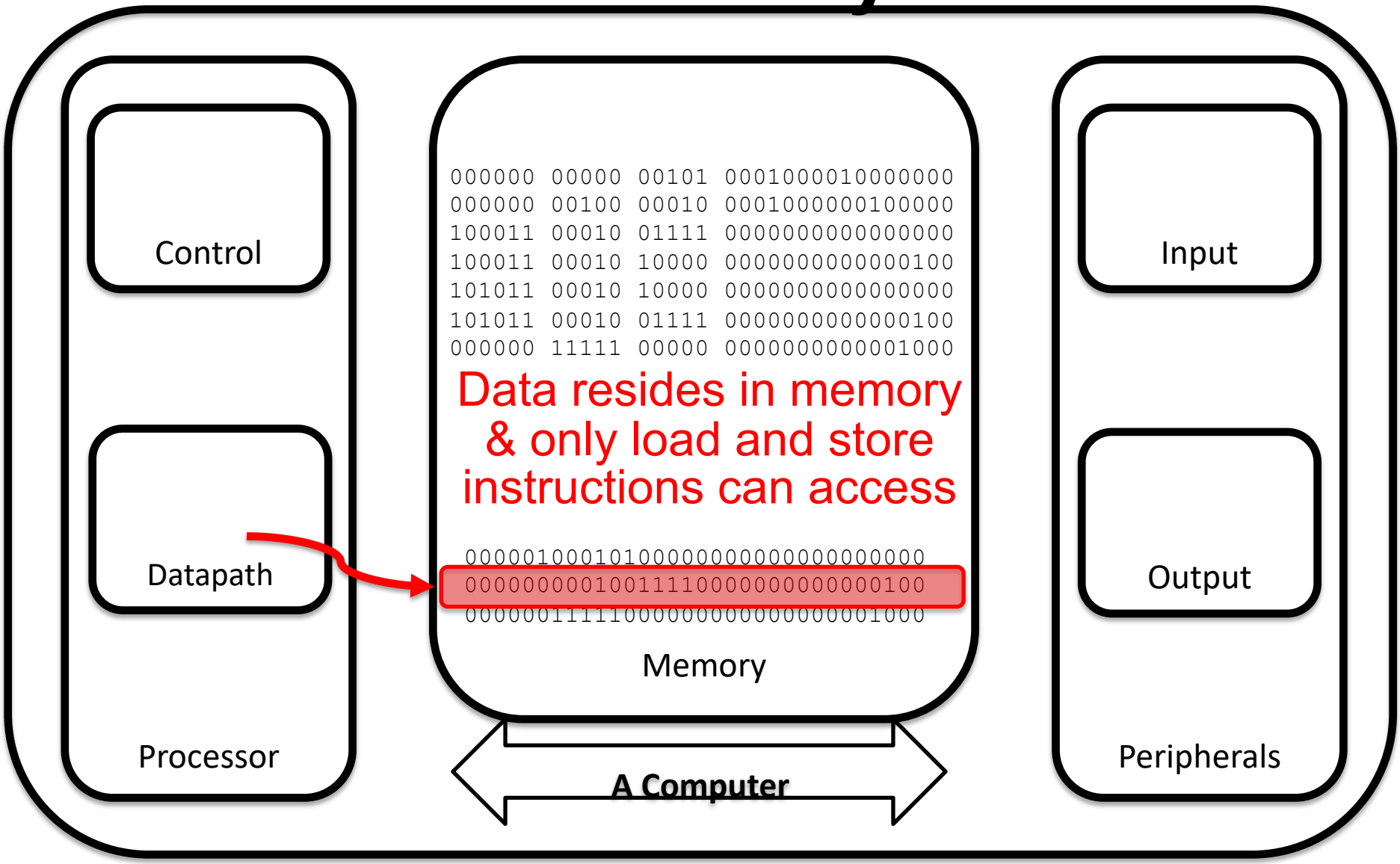
- What do these instructions do?
  - `addi $2, $1, 0`

MIPS Simple Arithmetic

Instruction	Example	Meaning	Comments
add	`add $1,$2,$3`	`$1 = $2 + $3`	3 operands; Overflow
subtract	`sub $1,$2,$3`	`$1 = $2 - $3`	3 operands; Overflow
add immediate	`addi $1,$2,100`	`$1 = $2 + 100`	+ constant; Overflow
add unsigned	`addu $1,$2,$3`	`$1 = $2 + $3`	3 operands; No overflow
sub unsigned	`subu $1,$2,$3`	`$1 = $2 - $3`	3 operands; No overflow
add imm unsign	`addiu $1,$2,100`	`$1 = $2 + 100`	+ constant; No overflow

- Your task: check out logical and shift instructions

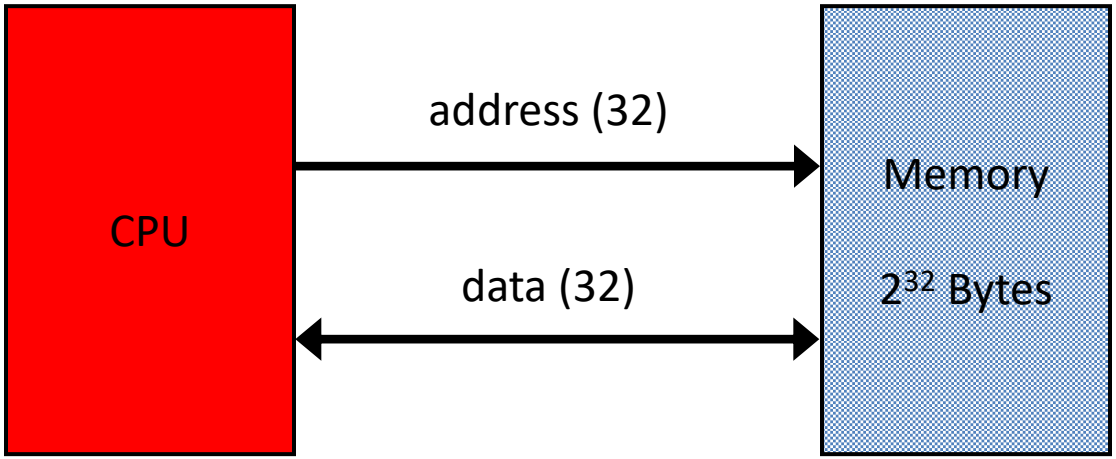
Data Stored in Memory



Memory

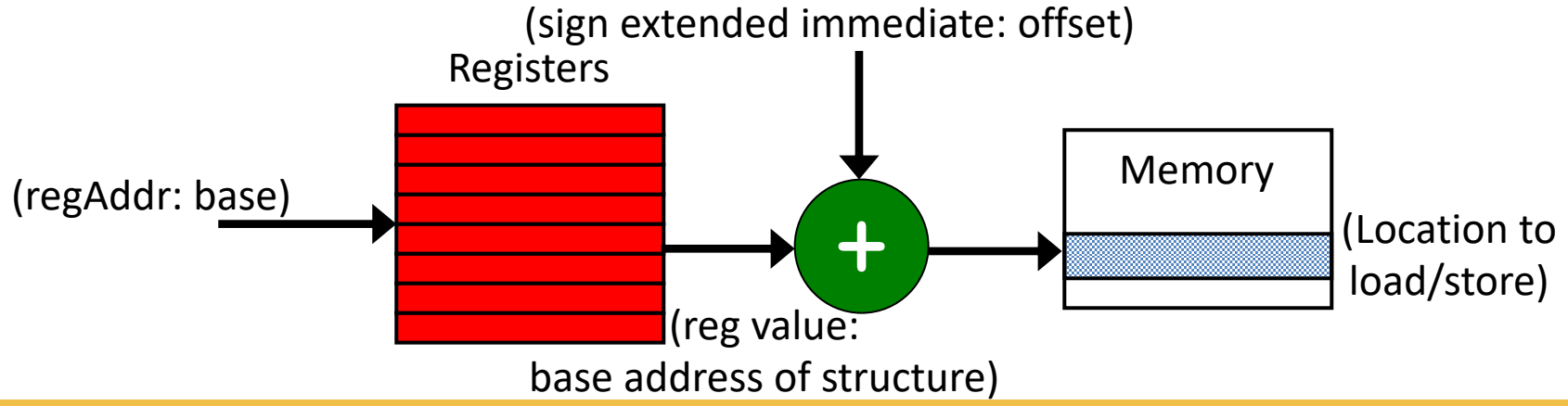
Memory Data Transfer

- Data transfer instructions are used to move data to and from memory
  - A **load** operation moves data from a memory location to a register
  - A **store** operation moves data from a register to a memory location



Memory Access

- All memory access happens through loads and stores
- Aligned words, half-words, and bytes
- Floating point loads and stores for accessing FP registers
- Displacement based addressing mode:



Data Transfer Instructions: Loads

- Data transfer instructions have three parts
  - Operator name (transfer size)
  - Destination register
  - Base address register and constant offset

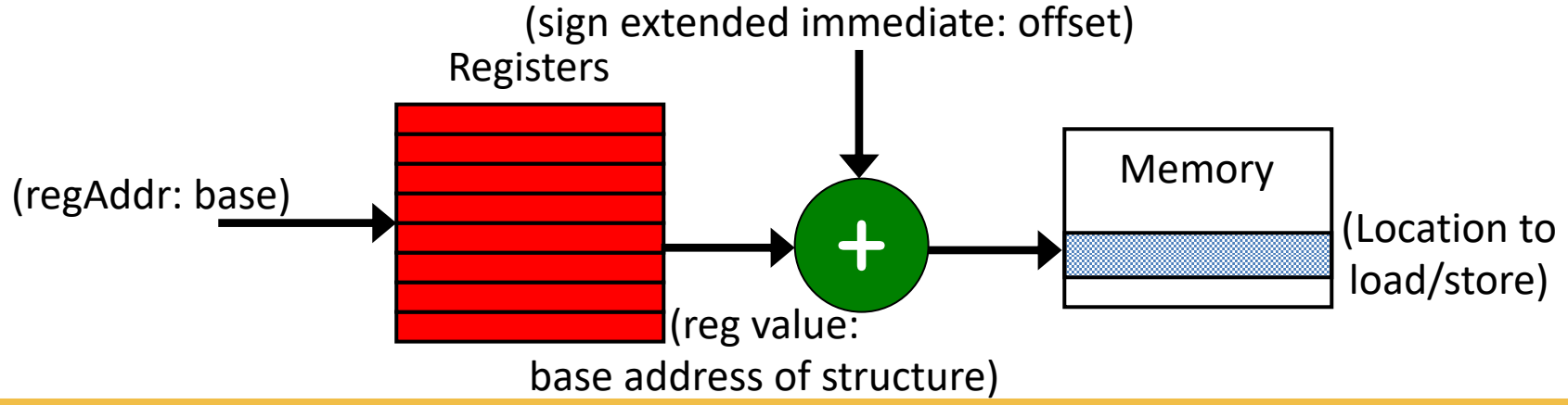
```
lw dst, offset(base)
```

  - Offset value is a **signed** constant

Memory Access

- All memory access happens through loads and stores
- Aligned words, half-words, and bytes
- Floating point loads and stores for accessing FP registers
- Displacement based addressing mode:

```
lw dst, offset(base)
```



Loading Data Example

- Consider the example

```
a = b + *c;
```

- Use the `lw` instruction to load
  - Assume **a=**`$s0`, **b=**`$s1`, **c=**`$s2`

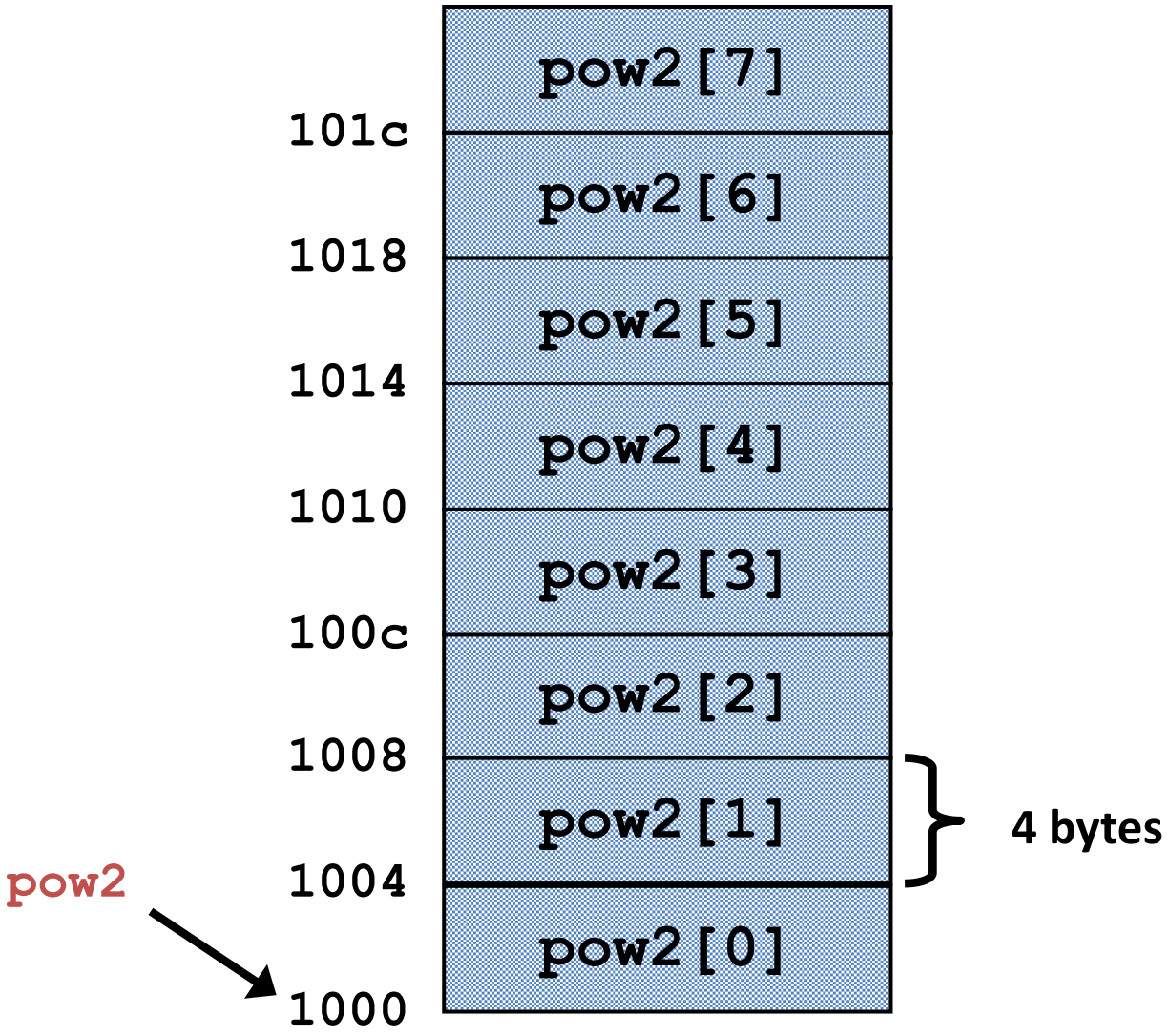
```
lw $t0, 0($s2)        # $t0 = Memory[c]
add $s0, $s1, $t0     # a = b + *c
```

Accessing Arrays

- Arrays are really pointers to the base address in memory
  - Address of element A[0]
- Use offset value to indicate which index
- Remember that addresses are in bytes, so multiply by the size of the element
  - Consider the integer array where **`pow2`** is the base address
  - With this compiler on this architecture, each int requires 4 bytes
  - The data to be accessed is at index 5: **`pow2[5]`**
  - Then the address from memory is **`pow2 + 5 * 4`**

- Unlike C, assembly does not handle pointer arithmetic for you!

Array Memory Diagram



Array Example

- Consider the example

```
a = b + pow2[7];
```

- Use the `lw` instruction offset
  - Assume `$s3` = 1000_{16}

```
lw $t0, 28($s3)    # $t0 = Mem[pow2[7]]
add $s0, $s1, $t0 # a = b + pow2[7]
```

C Code Example

Simple C procedure: sum_pow2=2^{b+c}

```
 1: int sum_pow2 (int b, int c)
 2: {
 3:   int pow2[8] = {1, 2, 4, 8, 16, 32, 64, 128};
 4:   int a, ret;
 5:   a = b + c;
 6:   if (a < 8)
 7:      ret = pow2[a];
 8:   else
 9:      ret = 0;
10:   return(ret);
11: }
```

More Complex Array Example

- Consider line 7 from sum_pow2()

```
ret = pow2[a];
```

- First find the correct offset
  - Again assume $s3 = 1000_{16}$

```
sll $t0, $s0, 2     # $t0 = 4 * a
add $t1, $s3, $t0   # $t1 = pow2 + 4*a
lw $v0 0($t1)       # $v0 = Mem[pow2[a]]
```

Storing Data

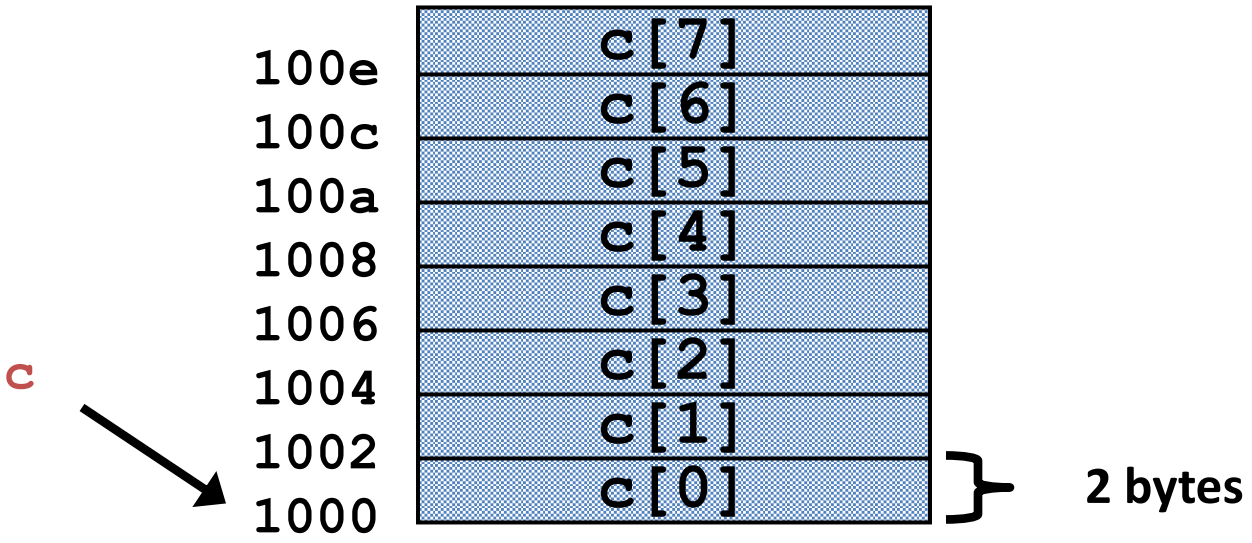
- Storing data is just the reverse and the instruction is nearly identical
- Use the **`sw`** instruction to copy a word from the source register to an address in memory

```
sw src, offset(base)
```

  - Offset value is a **signed** constant

A "short" Array Example

- ANSI C requires a short to be at least 16 bits and no longer than an int, but does not define the exact size
- For our purposes, treat a short as 2 bytes
- So, with a short array c[7] is at c + 7 * 2



MIPS Integer Load/Store

Instruction	Example	Meaning	Comments
store word	`sw $1,8($2)`	`Mem[8+$2]=$1`	Store word
store half	`sh $1,6($2)`	`Mem[6+$2]=$1`	Stores only lower 16b
store byte	`sb $1,5($2)`	`Mem[5+$2]=$1`	Stores only lowest byte
load word	`lw $1,8($2)`	`$1=Mem[8+$2]`	Load word
load halfword	`lh $1,6($2)`	`$1=Mem[6+$2]`	Load half; sign extend
load half unsign	`lhu $1,6($2)`	`$1=Mem[6+$2]`	Load half; zero extend
load byte	`lb $1,5($2)`	`$1=Mem[5+$2]`	Load byte; sign extend
load byte unsign	`lbu $1,5($2)`	`$1=Mem[5+$2]`	Load byte; zero extend

MIPS Integer Load/Store

In-class Assessment!

Access Code: DriveSafe

Note: sharing access code to those outside of classroom or using access while outside of classroom is considered cheating

In			
store			
store			
store			
load word	`lw $1,8($2)`	`$1=Mem[8+$2]`	Load word
load halfword	`lh $1,6($2)`	`$1=Mem[6+$2]`	Load half; sign extend
load half unsign	`lhu $1,6($2)`	`$1=Mem[6+$2]`	Load half; zero extend
load byte	`lb $1,5($2)`	`$1=Mem[5+$2]`	Load byte; sign extend
load byte unsign	`lbu $1,5($2)`	`$1=Mem[5+$2]`	Load byte; zero extend

In-Class Assessment

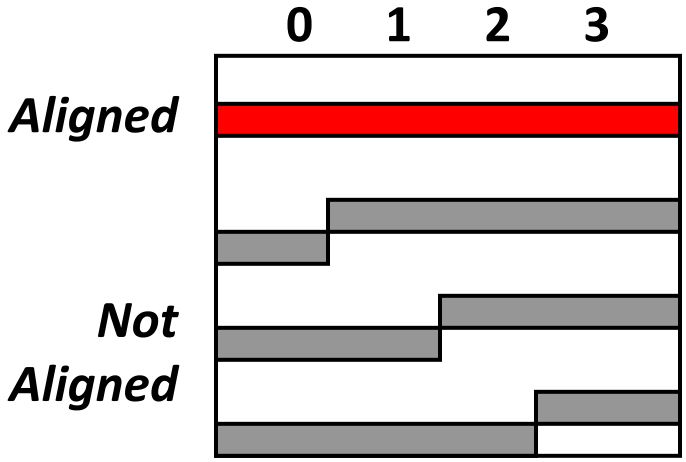
- What is the value contained in register $3 after the execution of the following code?

```
addi $1, $0, 10
addi $2, $0, 2002
addi $3, $0, 333
sw $2, 2($2)
addi $2, $2, 6
lw $3, -4($2)
```

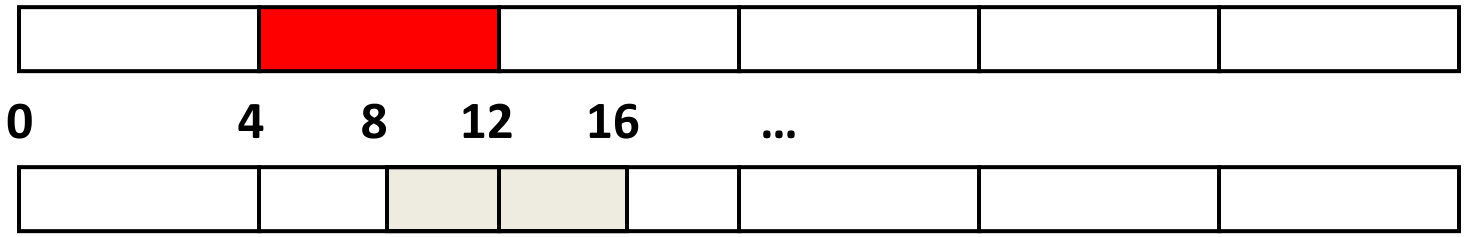
Memory

Alignment Restrictions

- In MIPS, data is required to fall on addresses that are multiples of the data size



- Consider word (4 byte) memory access



Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)