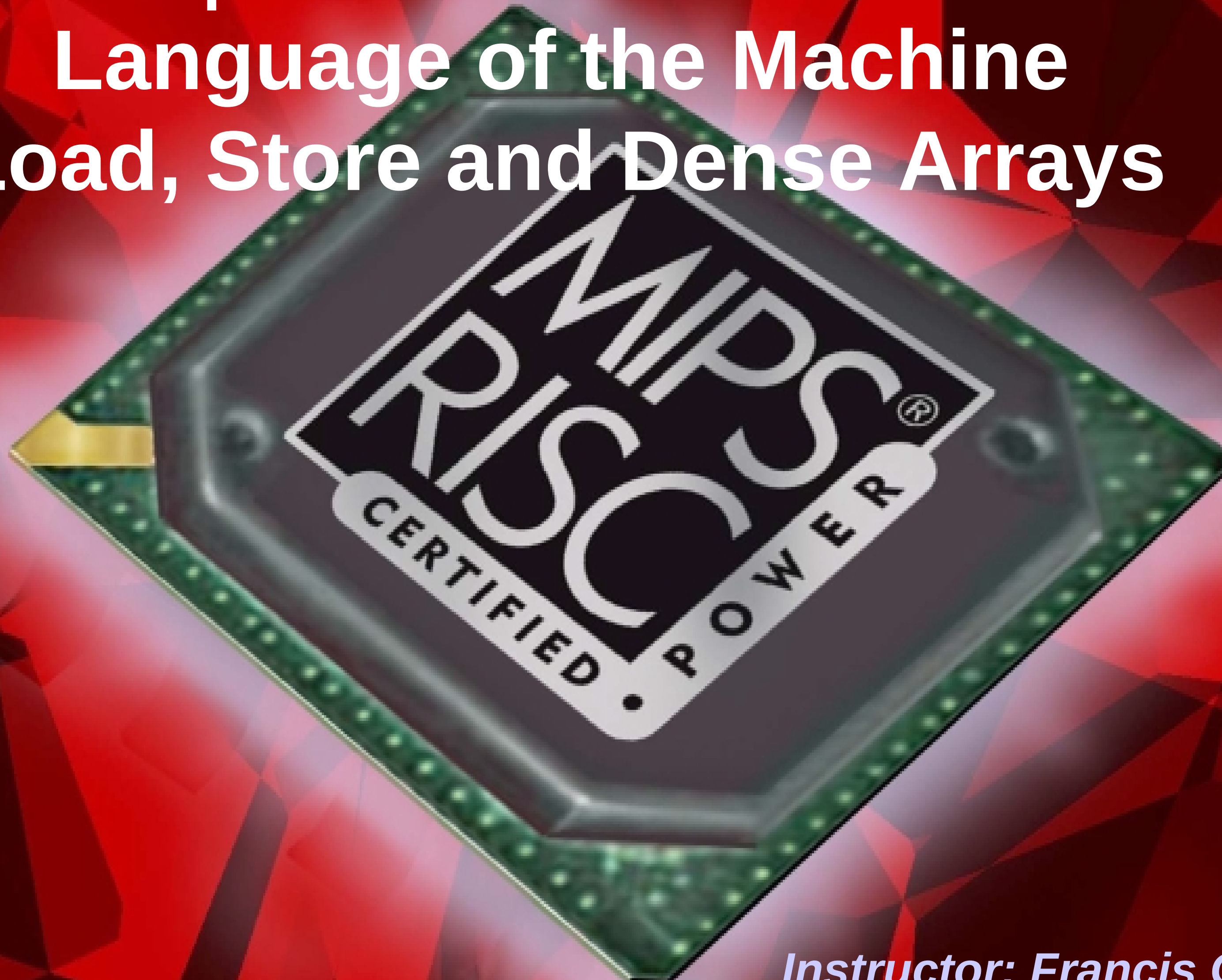
EECS 322 Computer Architecture

Language of the Machine
Load, Store and Dense Arrays



Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

Review: Design Abstractions

High Level Language
Program (e.g., C)

Compiler

Assembly Language
Program (e.g. MIPS)

Assembler

Machine Language
Program (MIPS)

Machine Interpretation

Control Signal
Specification

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

```
lw    $t0,   0($2)
lw    $t1,   4($2)
sw    $t1,   0($2)
sw    $t0,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

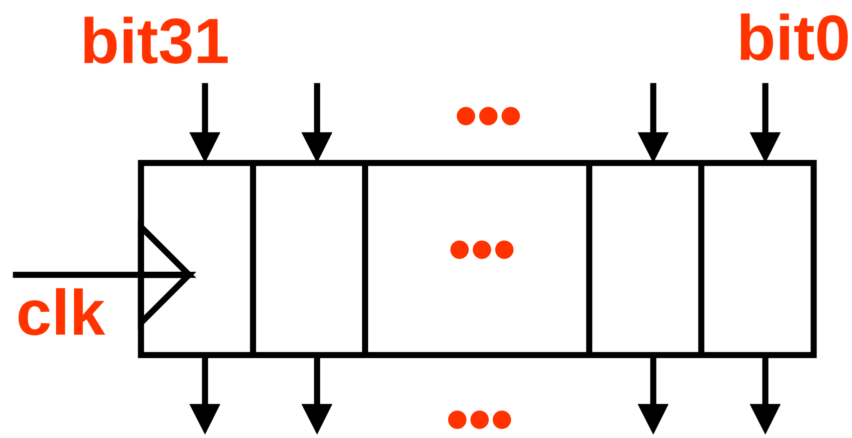
ALUOP[0:3] <= InstReg[9:11] & MASK

An abstraction omits unneeded detail, helps us cope with complexity

Review: Registers

- Unlike C++, assembly instructions cannot directly use variables. Why not? Keep Hardware Simple

- Instruction operands are registers: limited number of special locations; 32 registers in MIPS ($r0 - $r31)



Why 32? Performance issues: Smaller is faster

- Each MIPS register is 32 bits wide
  Groups of 32 bits called a word in MIPS

- A word is the natural size of the host machine.

Register Organization

- **Viewed as a**
  tiny single-dimension array (32 words),
  with an register address.

- **A register address ($r0-$r31) is**
  an index into the array

$r0	0	**32 bits of data**
$r1	1	**32 bits of data**
$r2	2	**32 bits of data**
$r3	3	**32 bits of data**
. . .	. . .	
$r28	28	**32 bits of data**
$r29	29	**32 bits of data**
$r30	30	**32 bits of data**
$r31	31	**32 bits of data**

ANSI C integers (section A4.2 Basic Types)

- **Examples: short x; int y; long z; unsigned int f;**
- **Plain int objects have the natural size suggested by the host machine architecture;**
- **the other sizes are provided to meet special needs**
- **Longer integers provide at least as much as shorter ones,**
- **but the implementation may make plain integers equivalent to either short integers, or long integers.**
- **The int types all represent signed values unless specified otherwise.**

Review: Compilation using Registers

- **Compile by hand using registers:**

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

Note: whereas C declares its operands, Assembly operands (registers) are fixed and not declared

- **Assign MIPS registers:**

```
# $s0=int f, $s1=int g, $s2=int h,
# $s3=int i, $s4=int j
```

- **MIPS Instructions:**

```
add $s0,$s1,$s2       # $s0 = g+h
add $t1,$s3,$s4       # $t1 = i+j
sub $s0,$s0,$t1       # f=(g+h)-(i+j)
```

ANSI C register storage class (section A4.1)

- Objects declared ***register*** are automatic, and ***(if possible)*** stored in fast registers of the machine.

- Previous example:
  register int f, g, h, i, j;
  f = (g + h) - (i + j);

If your variables exceed your number of registers, then not possible

- The register keyword tells the compiler your intent.

- This allows the programmer to guide the compiler for better results. (i.e. faster graphics algorithm)

- This is one reason that the C language is successful because it caters to the hardware architecture!

Assembly Operands: Memory

- C variables map onto registers
- What about data structures like arrays?
- But MIPS arithmetic instructions
  only operate on registers?
- Data transfer instructions
  transfer data between registers and memory

Think of memory as a large single dimensioned array, starting at 0

Memory Organization: bytes

- Viewed as a
  large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the
  index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

- C Language:
  - bytes multiple of word
  - Not guaranteed though

```
char f;
unsigned char g;
signed char h;
```

Memory Organization: words

- **Bytes are nice,**
  but most data items use larger "words"

- **For MIPS, a word is 32 bits or 4 bytes.**

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

. . .

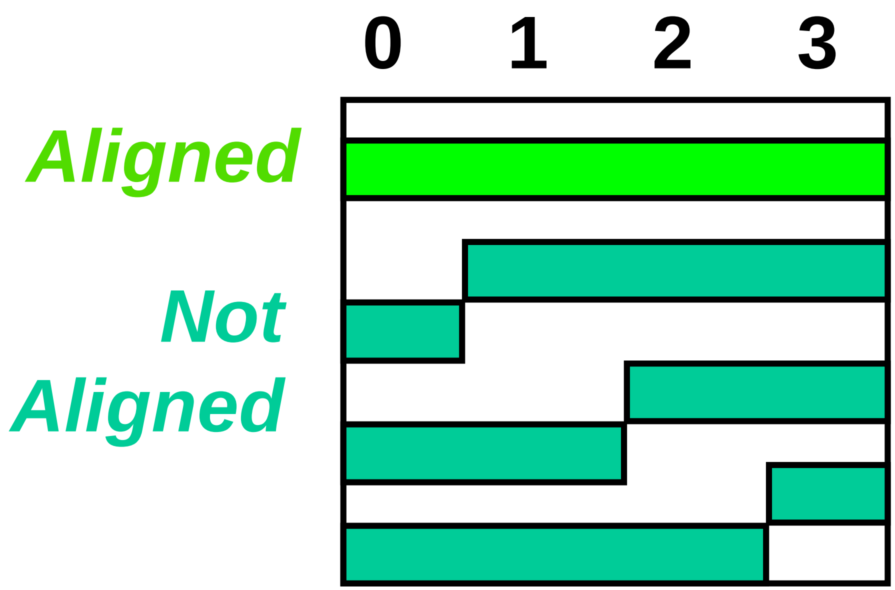
Note: Registers hold 32 bits of data
= word size (not by accident)

- **2^{32} bytes with byte addresses from 0 to 2^{32}-1**

- **2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4**

Memory Organization: alignment

• MIPS requires that all words start at addresses that are multiples of 4

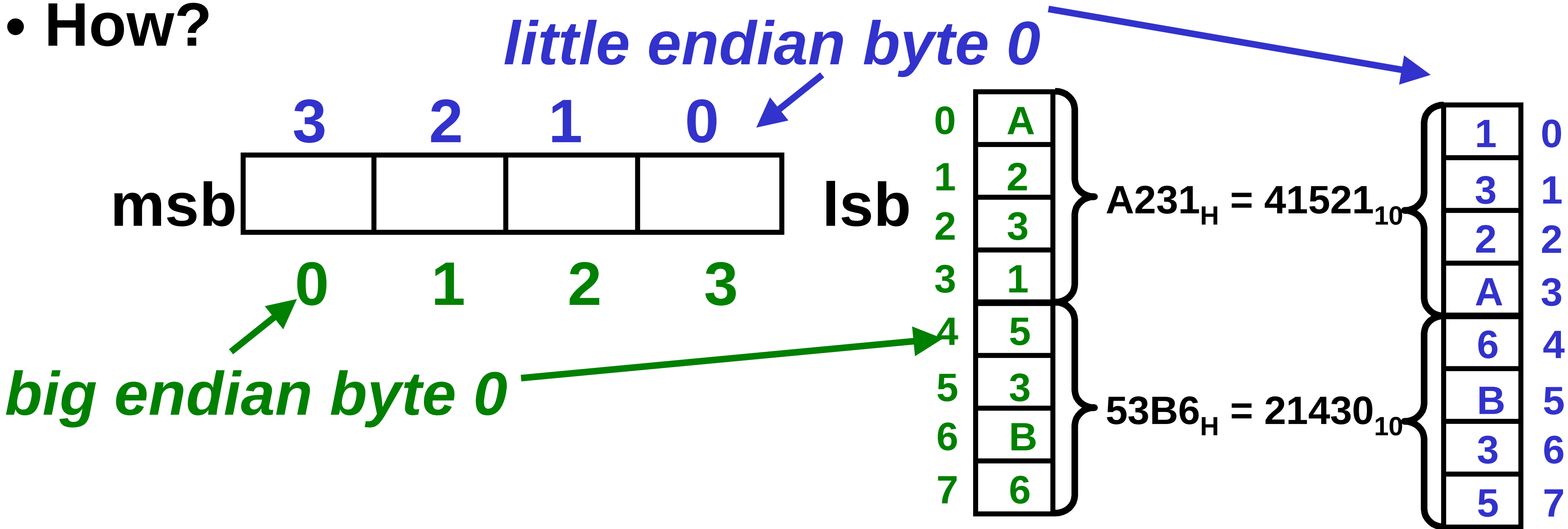


• Called alignment: objects must fall on address that is multiple of their size.

• (Later we'll see how alignment helps performance)

Memory Organization: Endian

- Words are aligned (i.e. 0,4,8,12,16,… not 1,5,9,13,…) i.e., what are the least 2 significant bits of a word address? Selects the which byte within the word
- How?



- **Little Endian** address of least significant byte: Intel 80x86, DEC Alpha
- **Big Endian** address of most significant byte: HP PA, IBM/Motorola PowerPC, SGI, Sparc

Data Transfer Instruction: Load Memory to Reg (lw)

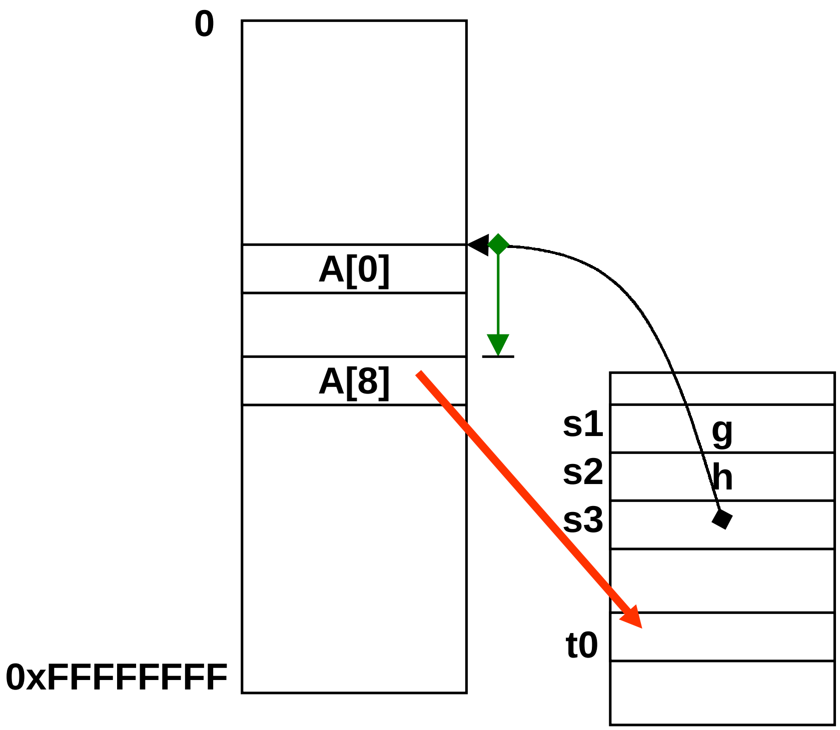
- **Load: moves a word from memory to register**
- **MIPS *syntax*, lw for load word:**
  - **operation name**
    - **register to be loaded**
      - **constant and register to access memory**

example: lw $t0, 8($s3)

Called "offset" **Called "base register"**

- **MIPS lw *semantics*:** reg[$t0] = Memory[8 + reg[$s3]]

lw example



- The value in register $s3 is an address
- Think of it as a pointer into memory

Suppose:

Array A address = 3000

reg[$s3]=Array A

reg[$t0]=12;

mem[3008]=42;

Then

lw $t0,8($s3)

Adds offset "8"

to $s3 to select A[8],

to put "42" into $t0

reg[$t0]=mem[8+reg[$s3]]

=mem[8+3000]=mem[3008]

=42 *=Hitchhikers Guide to the Galaxy*

Data Transfer Instruction: Store Reg to Memory (sw)

- **Store Word (sw):** **moves a word from register to memory**

- **MIPS *syntax*:** **sw $rt, offset($rindex)**
- **MIPS *semantics*:** mem[offset + reg[$rindex]] = reg[$rt]

- **MIPS *syntax*:** **lw $rt, offset($rindex)**
- **MIPS *semantics*:** reg[$rt] = mem[offset + reg[$rindex]]

- **MIPS *syntax*:** **add $rd, $rs, $rt**
- **MIPS *semantics*:** reg[$rd] = reg[$rs]+reg[$rt]

- **MIPS *syntax*:** **sub $rd, $rs, $rt**
- **MIPS *semantics*:** reg[$rd] = reg[$rs]-reg[$rt]

Compile Array Example

C code fragment:

```
register int g, h, i;
int     A[66];     /* 66 total elements: A[0..65] */
g = h + A[i];     /* note: i=5 means 6rd element */
```

Compiled MIPS assembly instructions:

```
add   $t1,$s4,$s4        # $t1 = 2*i
add   $t1,$t1,$t1        # $t1 = 4*i
add   $t1,$t1,$s3        #$t1=addr A[i]
lw    $t0,0($t1)         # $t0 = A[i]
add   $s1,$s2,$t0        # g = h + A[i]
```

Execution Array Example: g = h + A[i];

C variables	g	h	A	i		
Instruction	$s1	$s2	$s3	$s4	$t0	$t1
suppose (mem[3020]=38)	?	4	3000	5	?	?
add $t1,$s4,$s4	?	4	3000	5	?	?
add $t1,$t1,$t1	?	4	3000	5	?	10
add $t1,$t1,$s3	?	4	3000	5	?	20
lw $t0,0($t1)	?	4	3000	5	?	3020
add $s1,$s2,$t0	?	4	3000	5	38	20
??? ?,?,?	42	4	3000	5	?	20

Immediate Constants

C expressions can have constants:

i = i + 10;

MIPS assembly code:

```
# Constants kept in memory with program
lw    $t0, 0($s0)        # load 10 from memory
add   $s3,$s3,$t0        # i = i + 10
```

MIPS using constants: (addi: add immediate)

So common operations, have instruction to add constants (called "immediate instructions")

```
addi $s3,$s3,10          # i = i + 10
```

Constants: Why?

Why include immediate instructions?

Design principle: Make the common case fast

Why faster?

a) Don't need to access memory
b) 2 instructions v. 1 instruction

Zero Constant

Also,perhaps most popular constant is zero.

MIPS designers reserved 1 of the 32 register to always have the value 0; called $r0, $0, or "<u>$zero</u>"

Useful in making additional operations from existing instructions;

copy registers: $s2 = $s1;

add $s2, $s1, $zero ***# $s2 = $s1 + 0***

2's complement: $s2 = –$s1;

sub $s2, $zero, $s1 ***# $s2 = – $s1***

Load a constant: $s2 = number;

addi $s2, $zero, 42 ***# $s2 = 42***

C Constants

C code fragment

```
int i;
const int limit = 10;

i = i + limit;
```

Is the same as

```
i = i + limit; /* but more readable */
```

And the compiler will protect you from doing this

```
limit=5;
```

Class Homework: Due next class

C code fragment:

```
register int g, h, i, k;
int    A[5], B[5];
B[k] = h + A[i+1];
```

1. Translate the C code fragment into MIPS

2. Execute the C code fragment using:

A=address 1000, B=address 5000, i=3, h=10, k=2,
int A[5]={24, 33, 76, 2, 19};
/* i.e. A[0]=24; A[1]=33; … */ .