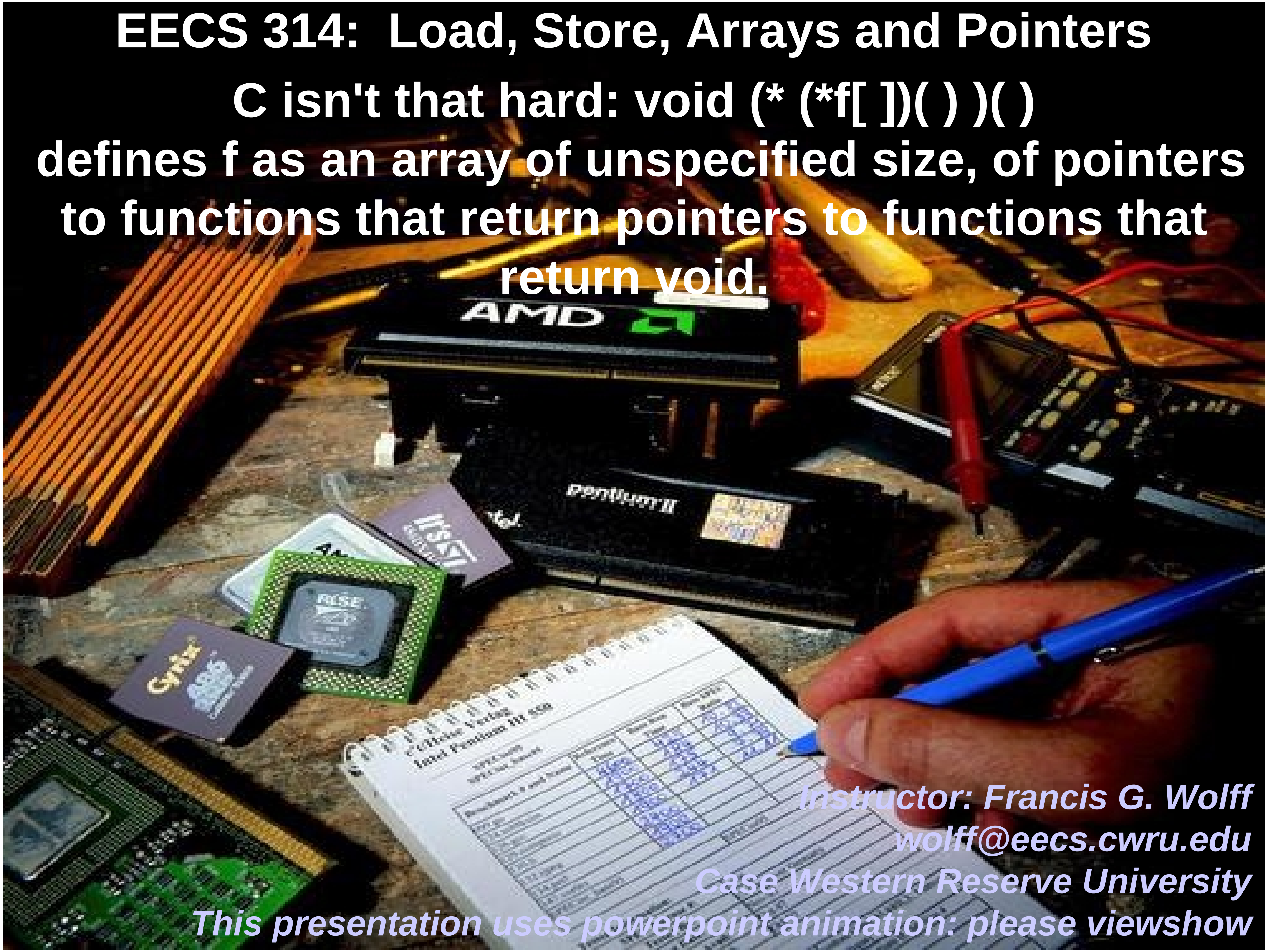
EECS 314: Load, Store, Arrays and Pointers

C isn't that hard: void (* (*f[ ])( ))( )

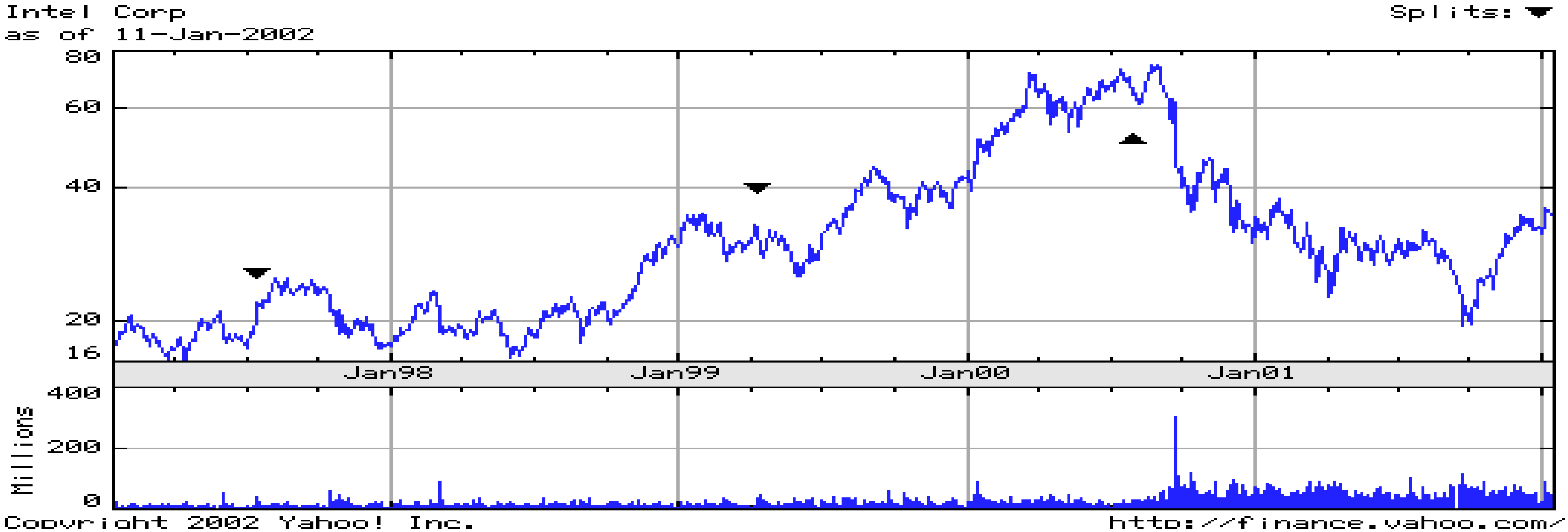
defines f as an array of unspecified size, of pointers to functions that return pointers to functions that return void.

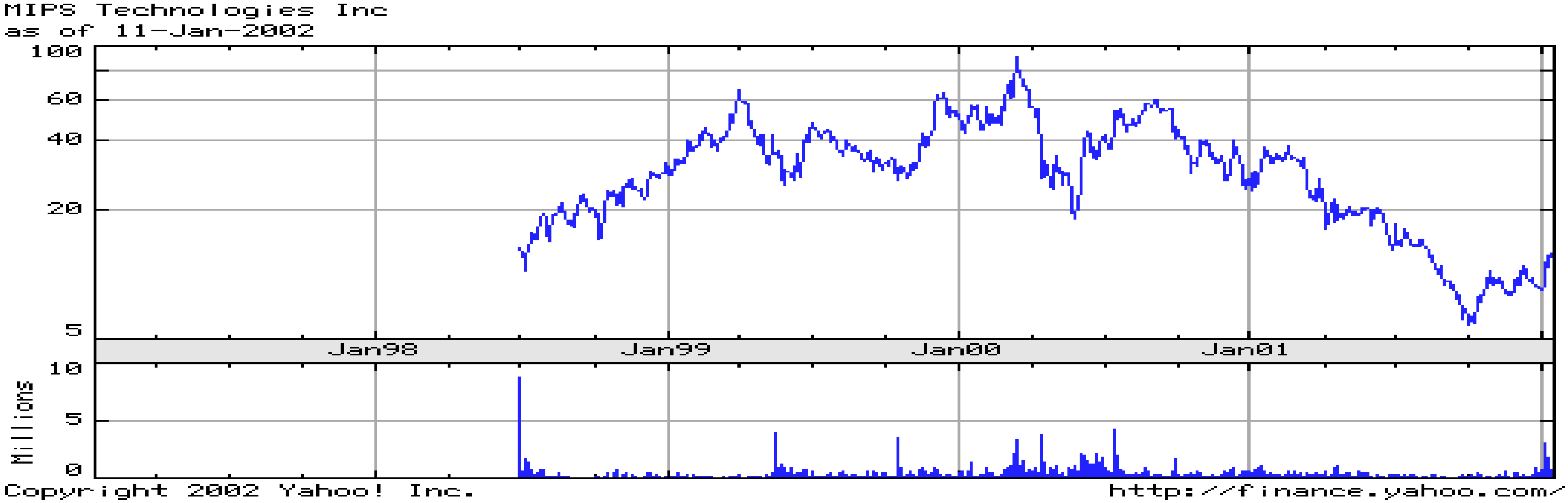


Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

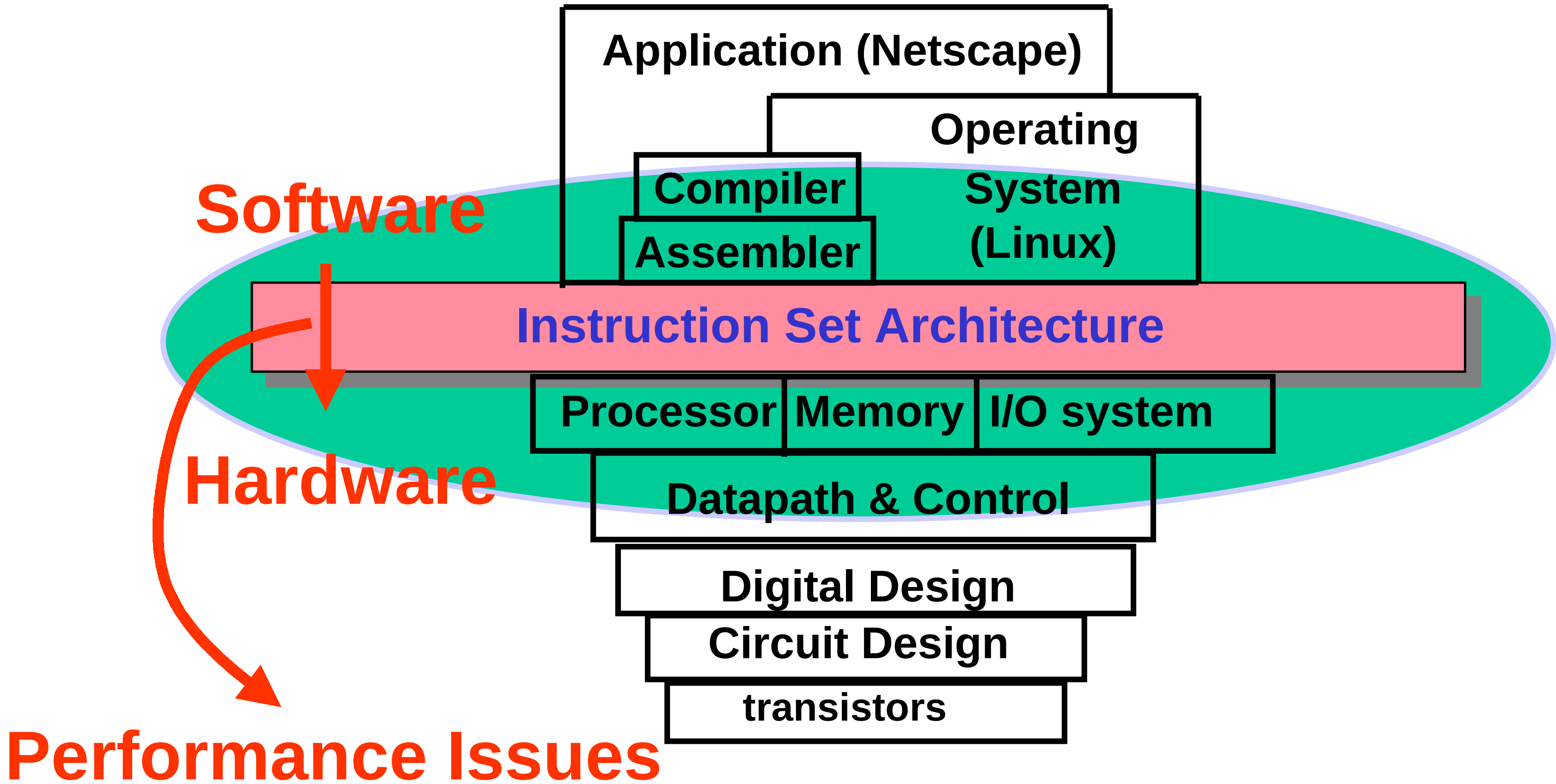
Compagnie du jour: Intel Corporation www.intel.com





Review: Design Abstractions

- Coordination of many *levels of abstraction*



Speed
Power
Size

Review: Design Abstractions

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

Compiler

Assembly Language
Program (e.g. MIPS)

Assembler

Machine Language
Program (MIPS)

Instruction Set Architecture

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

Register Organization

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

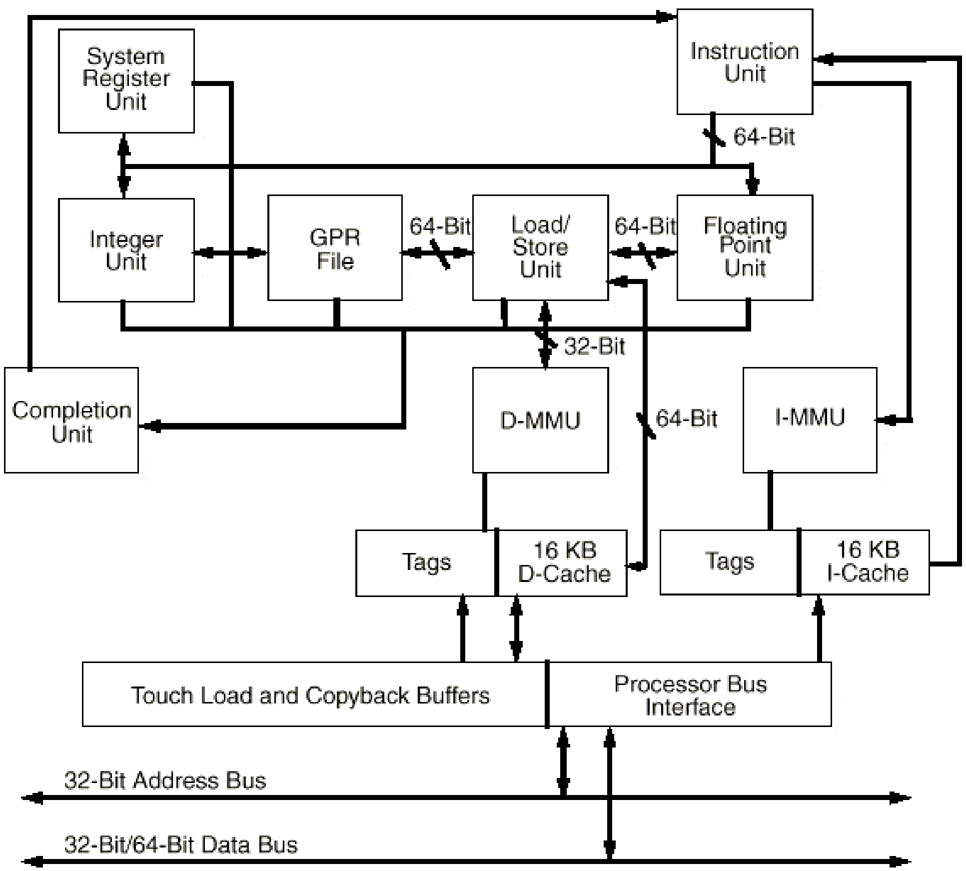
- **A register address ($r0-$r31) is an index into the array**
  - **register int r[32]; /* C notation */**

$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

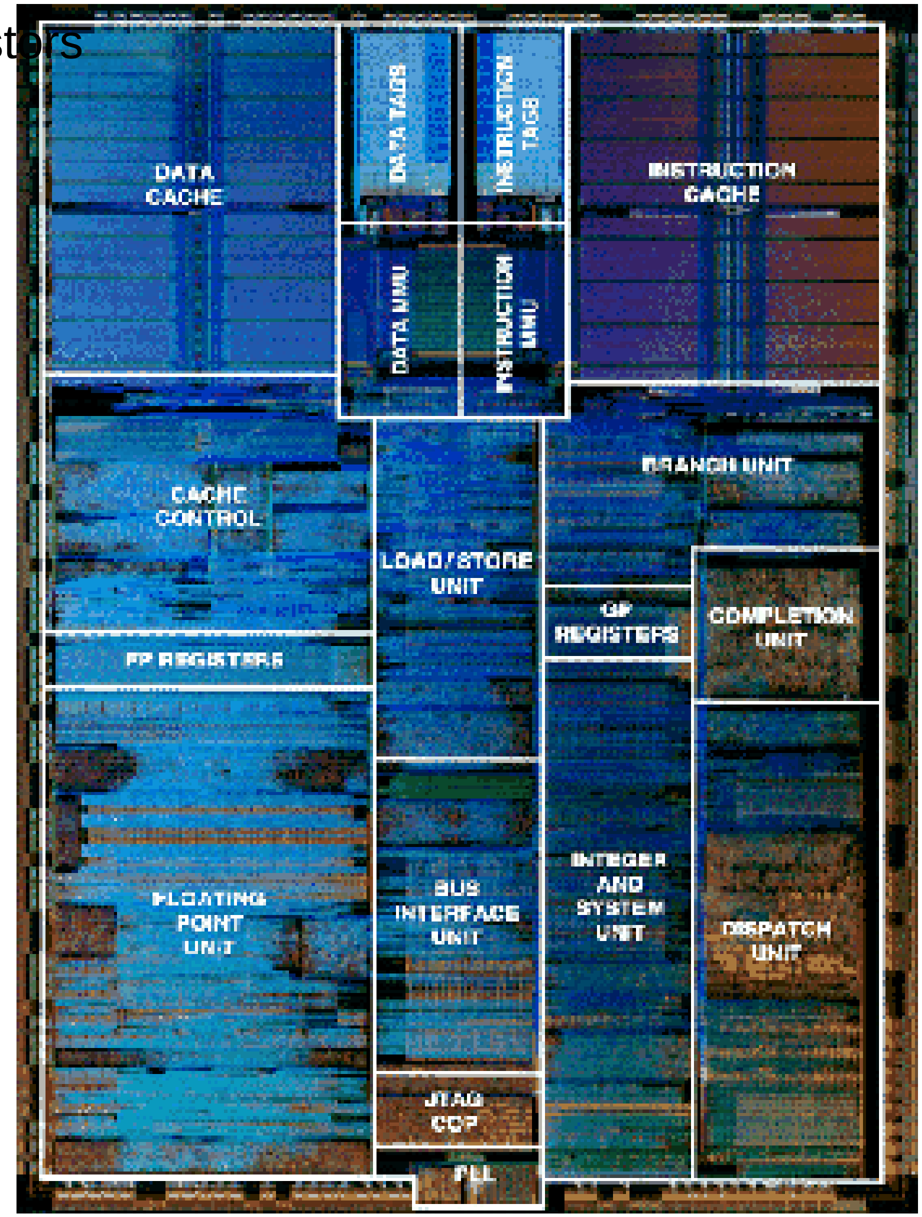
PowerPC 603: Registers

Year: 1994 / 66 to 80 MHz
Process: 0.5-micron CMOS / 1.6 million transistors
Cache: 8Kb Inst. / 8 kb Data

Year: 1997 / 225 Mhz to 300 Mhz
Process: 0.5 to 0.35-micron CMOS
Cache: 16 Kb Inst / 16 Kb Data



Motorola's PowerPC 603e™ RISC Microprocessor - 200 MHz



MIPS registers and conventions

Name	Number	Conventional usage
$0	0	Constant 0
$v0-$v1	2-3	Expression evaluation & function results
$a0-$a3	4-7	Arguments 1 to 4
$t1-$t9	8-15,24,35	Temporary (not preserved across call)
$s0-$s7	16-23	Saved Temporary (preserved across call)
$k0-$k1	26-27	Reserved for OS kernel
$gp	28	Pointer to global area
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address (used by function call)

MIPS Register Name translation

/* calculate (PH p. 109, file: simplecalc.s) */

```
register int g=5, h= -20, i=13, j=3, f;
f = (g + h) - (i + j);
```

Assember .s		Translated (1 to 1 mapping)		
addi	$s1, $0, 5	addi	$17, $0, 5	# g = 5
addi	$s2, $0, -20	addi	$18, $0, -20	# h = -20
addi	$s3, $0, 13	addi	$19, $0, -20	# i = 13
addi	$s4, $0, 3	addi	$20, $0, 3	# j = 3
add	$t0, $s1, $s2	add	$8, $17, $18	# $t0=g + h
add	$t1, $s3, $s4	add	$9, $19, $20	#$t1=i + j
sub	$s0, $t0, $t1	sub	$16, $8, $9	# f=(g+h)-(i+j)

SPIM: System call 1: print_int $a0

• System calls are used to interface with the operating system to provide <u>device independent</u> services.

• System call 1 converts the binary value in register $a0 into ascii and displays it on the console.

• This is equivalent in the C Language: printf("%d", $a0)

<u>Assember .s</u>		<u>Translated (1 to 1 mapping)</u>		
li	$v0, 1	ori	$2, $0, 1	#print_int (system call 1)
add	$a0,$0,$s0	add	$4,$0,$16	#put value to print in $a0
syscall		syscall		

SPIM: System Services

Service	Code	Arguments	Result
print_int	1	$a0=integer	
print_float	2	$f12=float	
print_double	3	$f12=double	
print_string	4	$a0=string	
read_int	5		$v0=integer
read_float	6		$f0=float
read_double	7		$f0=double
read_string	8	$a0=buf, $a1=len	
sbrk	9	$a0=amount	$v0=address
exit	10		

SPIM: System call 4: print_string $a0

- System call 4 copies the contents of memory located at $a0 to the console until a **zero** is encountered
- This is equivalent in the C Language: printf("%s", $a0)

Assember .s	Translated (1 to 1 mapping)
.data	
.globl msg3	
msg3: .asciiz "\nThe value of f is: "	
.text	
li $v0, 4	ori $2, $0, 4 #print_string
la $a0,msg3	lui $4,4097 #address of string
syscall	syscall

Note the "z" in asciiz

msg3 is just a label but must match

.asciiz data representations

.data: items are place in the data segment

which is not the same the same as the .text segment !

<u>Assember .s</u>

msg3: .asciiz “\nThe va”

<u>Same as in assembler.s</u>

msg3: .byte ‘\n’,’T’,’h’, ‘e’, ‘ ‘, ‘v’, ‘a’, 0

<u>Same as in assembler.s</u>

msg3: .byte 0x0a, 0x54, 0x68, 0x65

.byte 0x20, 0x76, 0x61, 0x00

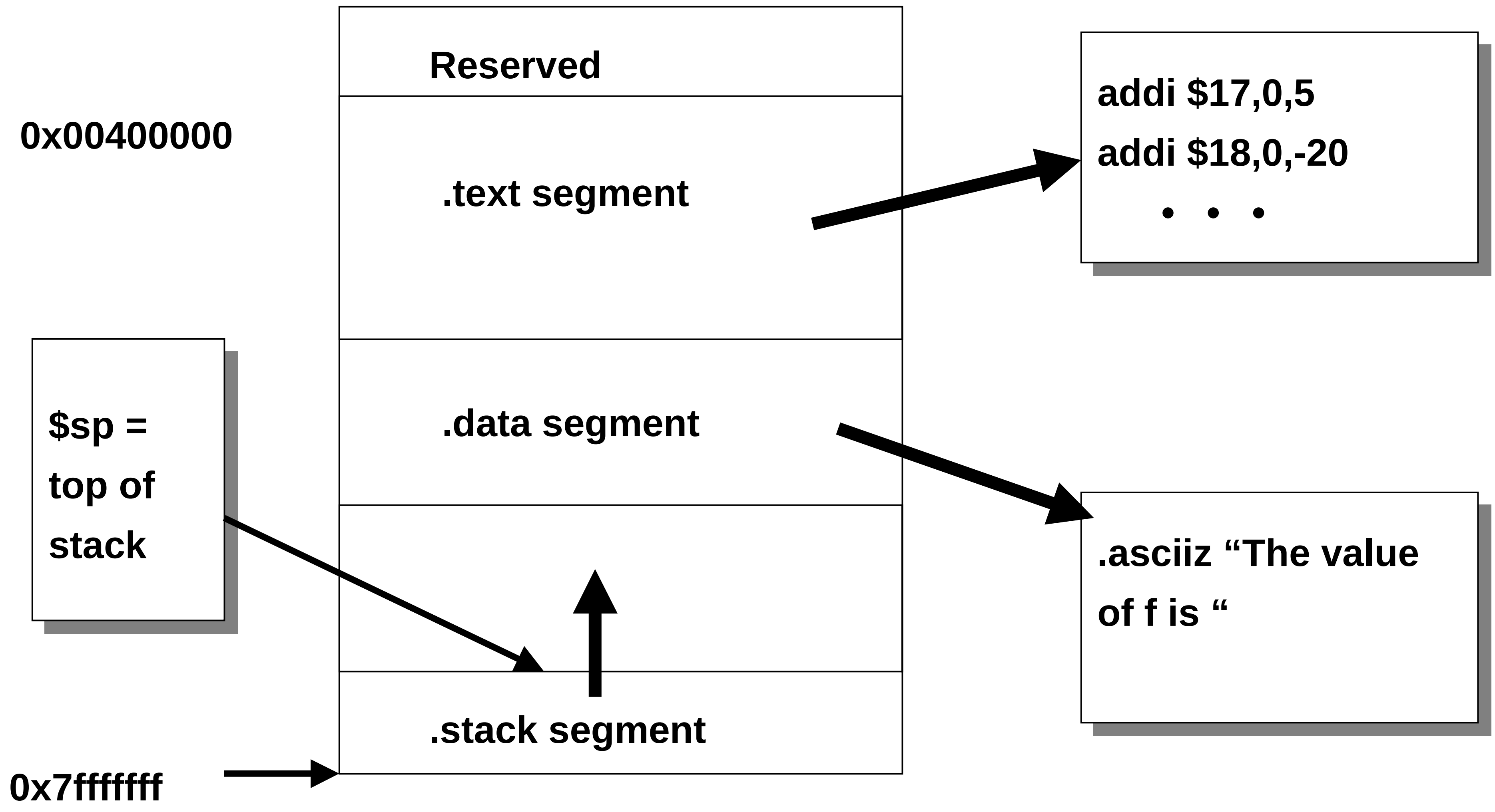
Big endian format

<u>Same as in assembler.s</u>

msg3: .word 0x6568540a, 0x00617620

Translated in the .data segment: 0x6568540a 0x00617620

Memory layout: segments



- Segments allow the operating system to protect memory
- Like Unix file systems: .text Execute only, .data R/W only

Hello, World: hello.s

Note: alternating .text, .data, .text

```
# main( ) {
#         printf(”\nHello World”\n”);
# }
          .globl main
main:                                 #main has to be a global label
          addu    $s7, $0, $ra        #save the return address in a global reg.
          .data
          .globl  hello
hello:    .asciiz "\nHello World\n"           #string to print
          .text
          li      $v0, 4                      # print_str (system call 4)
          la      $a0, hello                  # $a0=address of hello string
          syscall

          # Usual stuff at the end of the main
          addu    $ra, $0, $s7        #restore the return address
          jr      $ra                 #return to the main program
          add     $0, $0, $0          #nop
```

Simplecalc.s (PH p. 109)

Order of .text and .data not important

```
         .globl main
main:    addu    $s7, $0, $ra       #save the return address
         addi    $s1, $0, 5         #g = 5
         addi    $s2, $0, -20       #h = -20
         addi    $s3, $0, 13        #i = 13
         addi    $s4, $0, 3         #j = 3
         add     $t0, $s1, $s2      #register $t0 contains g + h
         add     $t1, $s3, $s4      #register $t1 contains i + j
         sub     $s0, $t0, $t1      #f = (g + h) - (i + j)
         li      $v0, 4             #print_str (system call 4)
         la      $a0, message       # address of string
         syscall
         li      $v0, 1             #print_int (system call 1)
         add     $a0, $0, $s0       #put value to print in $a0
         syscall
         addu    $ra, $0, $s7       #restore the return address
         jr      $ra                #return to the main program
         add     $0, $0, $0         #nop
         .data
         .globl  message
message:         .asciiz "\nThe value of f is: "      #string to print
```

Simplecalc.s without symbols (PH p. 109)

```
                .text
0x00400020      addu    $23, $0, $31        # addu   $s7, $0, $ra
0x00400024      addi    $17, $0, 5          # addi   $s1, $0, 5
0x00400028      addi    $18, $0, -20        # addi   $s2, $0, -20
0x0040002c      addi    $19, $0, 13         # addi   $s3, $0, 13
0x00400030      addi    $20, $0, 3          # addi   $s4, $0, 3
0x00400034      add     $8, $17, $18        # add    $t0, $s1, $s2
0x00400038      add     $9, $19, $20        # add    $t1, $s3, $s4
0x0040003c      sub     $16, $8, $9         # sub    $s0, $t0, $t1
0x00400040      ori     $2, 0, 4            #print_str (system call 4)
0x00400044      lui     $4, 0x10010000      # address of string
0x00400048      syscall
0x0040004c      ori     $2, 1               #print_int (system call 1)
0x00400050      add     $4, $0, $16         #put value to print in $a0
0x00400054      syscall
0x00400058      addu    $31, $0, $23        #restore the return address
0x0040005c      jr      $31                 #return to the main program
0x00400060      add     $0, $0, $0          #nop
                .data
0x10010000      .word   0x6568540a, 0x6c617620, 0x6f206575
                .word   0x20662066, 0x203a7369, 0x00000000
```

Single Stepping

Values changes after the instruction!

$pc	$t0 $8	$t1 $9	$s0 $16	$s1 $17	$s2 $18	$s3 $19	$s4 $20	$s7 $23	$ra $31
00400020	?	?	?	?	?	?	?	?	400018
00400024	?	?	?	?	?	?	?	400018	400018
00400028	?	?	?	5	?	?	?	400018	400018
0040002c	?	?	?	5	ffffffec	?	?	400018	400018
00400030	?	?	?	5	ffffffec	0d	?	400018	400018
00400034	?	?	?	5	ffffffec	0d	3	400018	400018
00400038	ffffff1	?	?	5	ffffffec	0d	?	400018	400018
0040003c	?	10	?	5	ffffffec	0d	?	400018	400018
00400040	?	?	ffffffe1	5	ffffffec	0d	?	400018	400018

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:

```
register 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**
	. . .

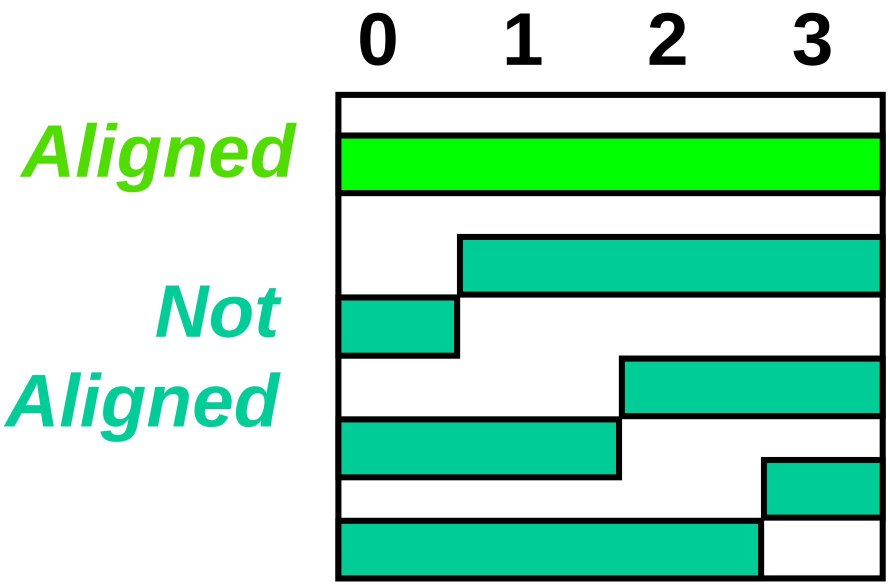
Note: Registers hold 32 bits of data
= word size (not by accident)

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

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

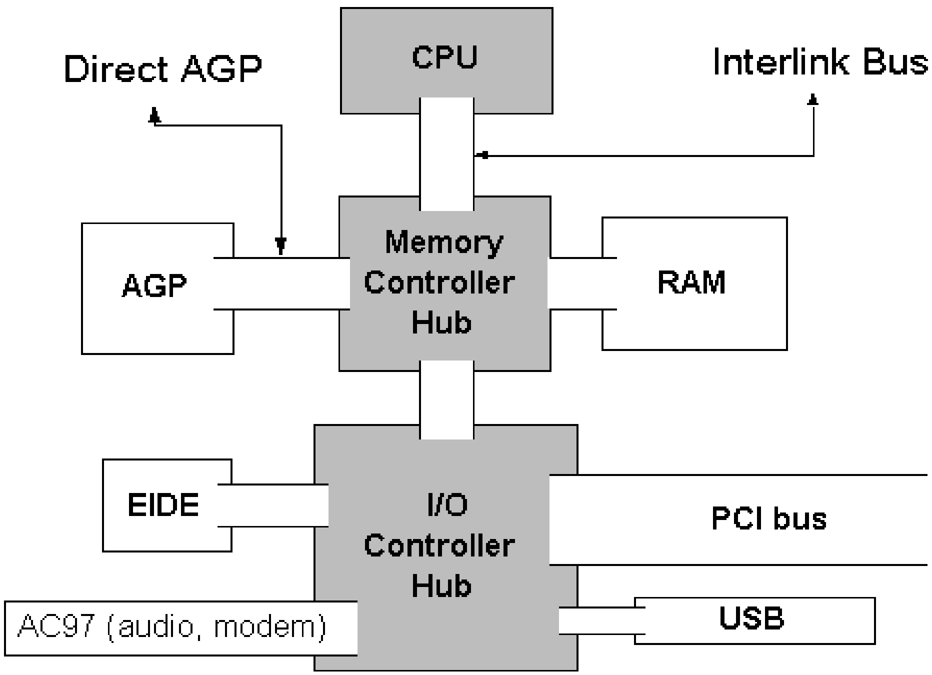
Memory Organization: alignment

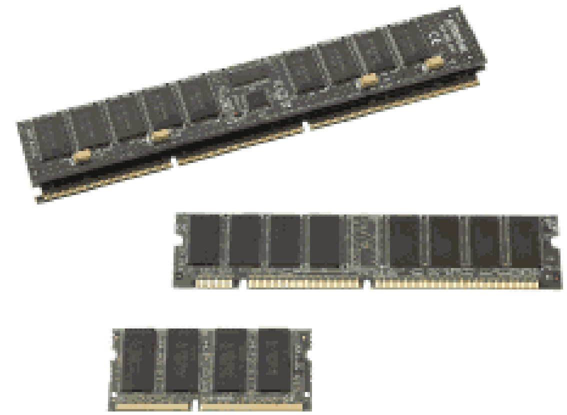
- **MIPS requires that all words start at addresses that are multiples of 4**

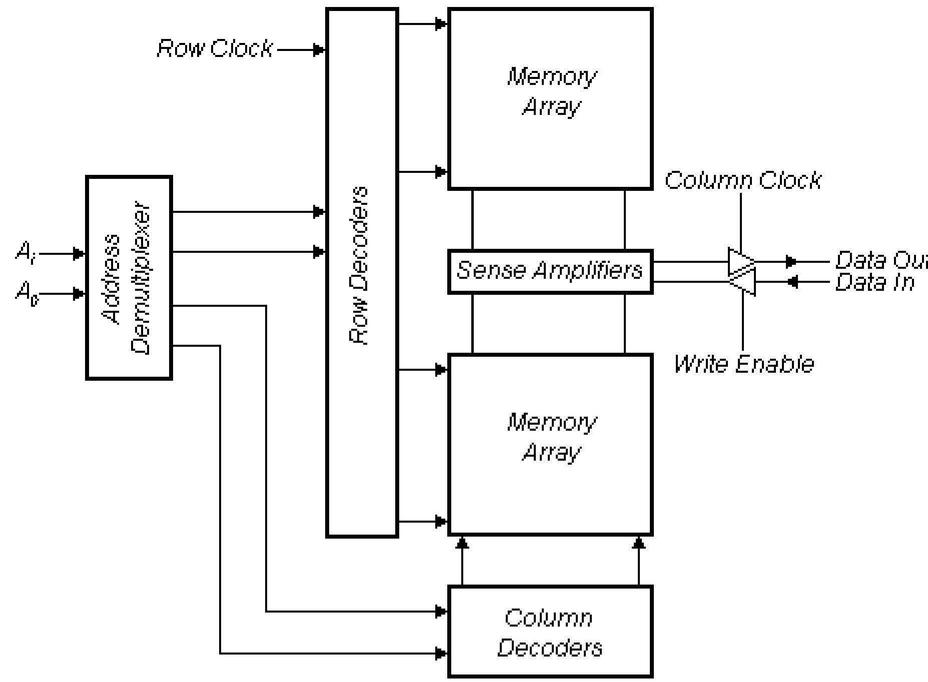


- **Called alignment: objects must fall on address that is multiple of their size.**
- **(Later we'll see how alignment helps performance)**

Memory: Bus

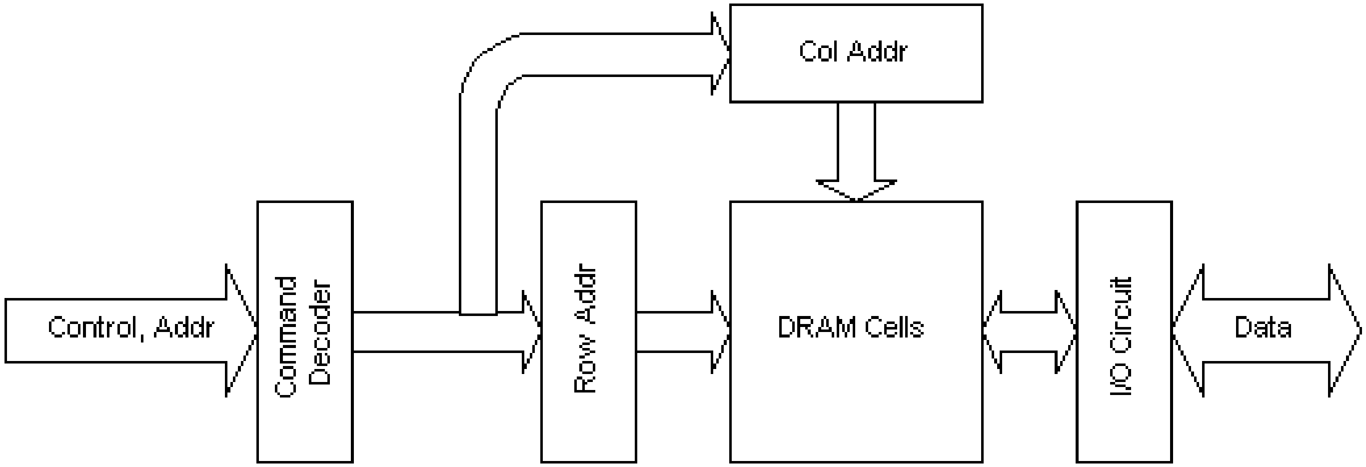




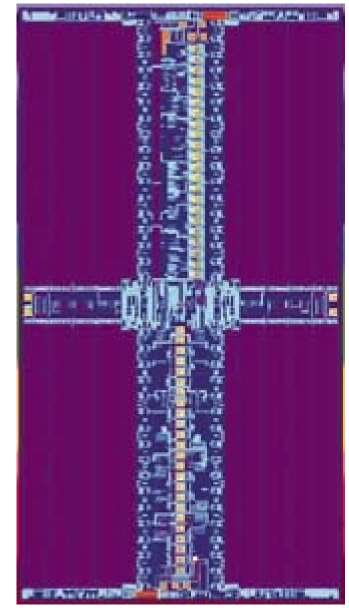


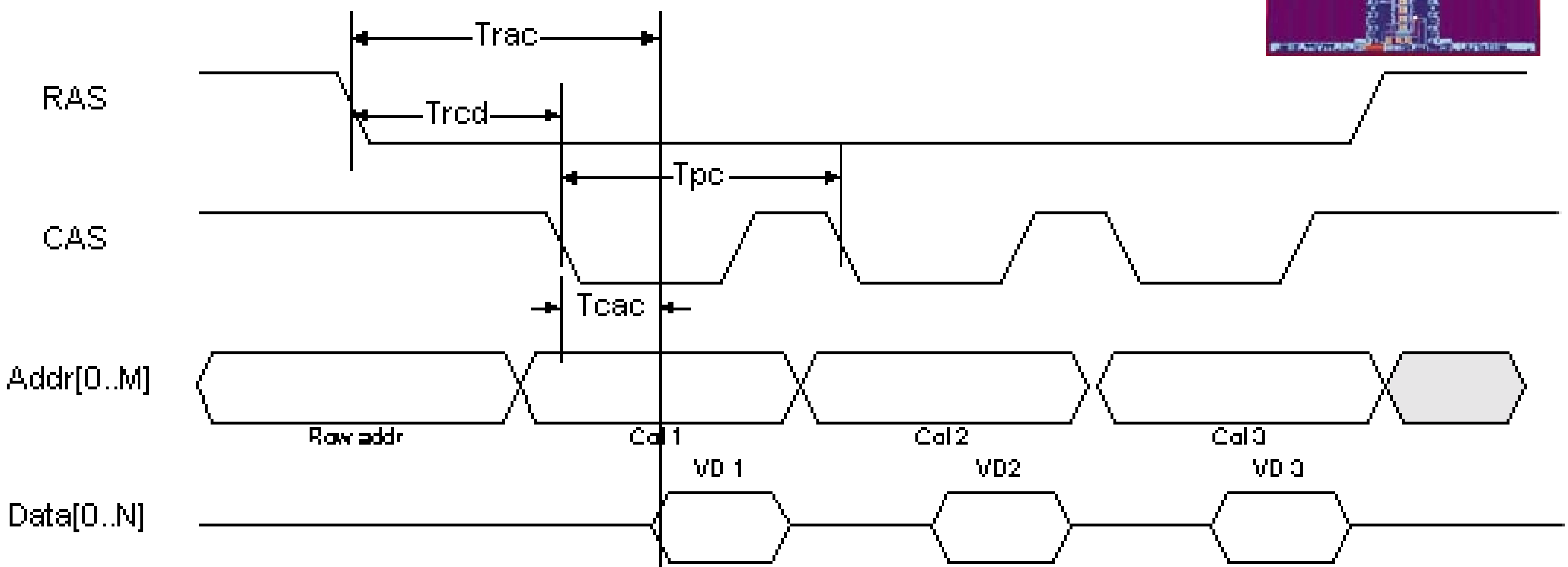
Internal Architecture of a DRAM

Memory: Dram Timing



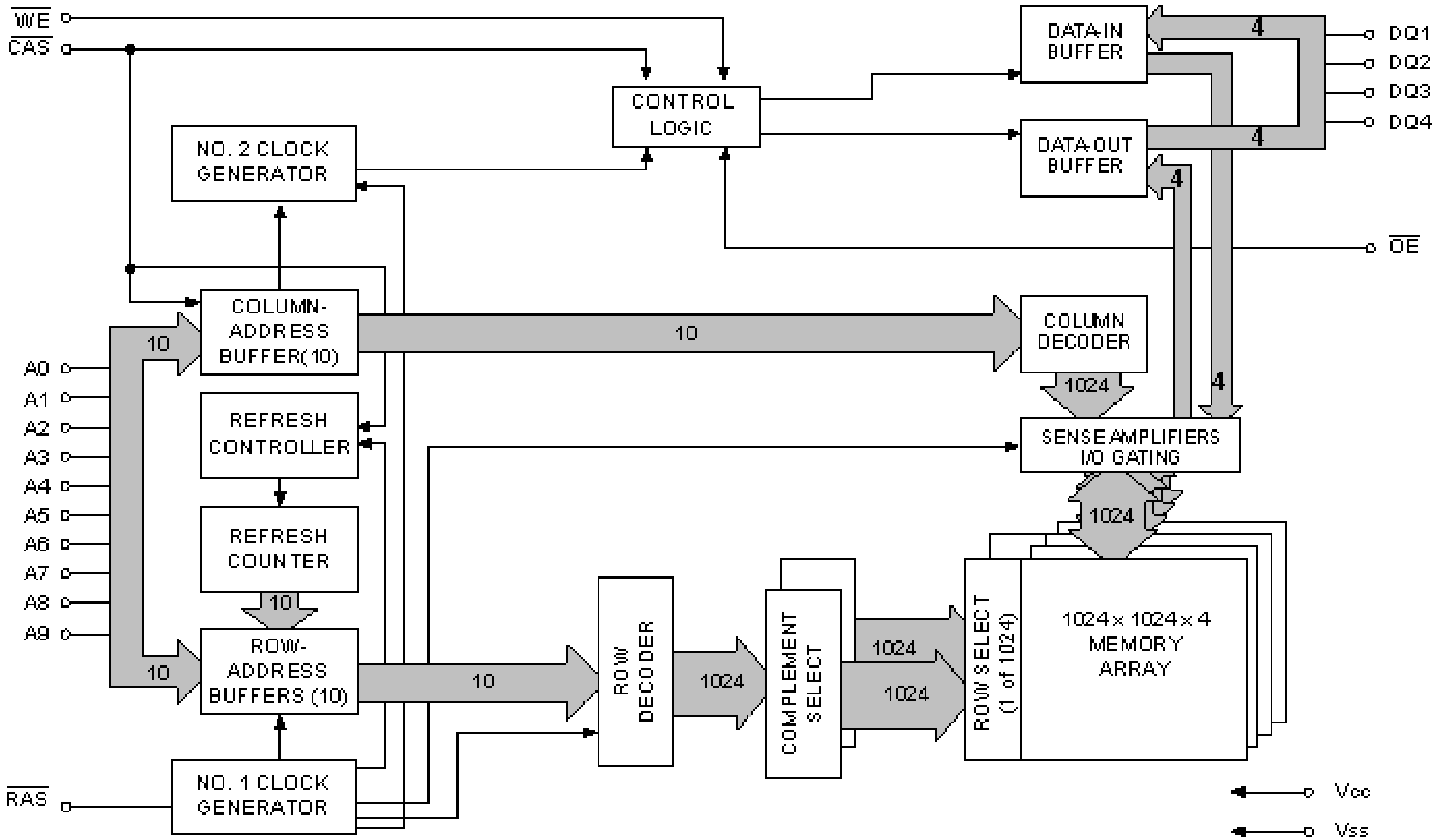
DRAM Structure





FPM/EDO read timing

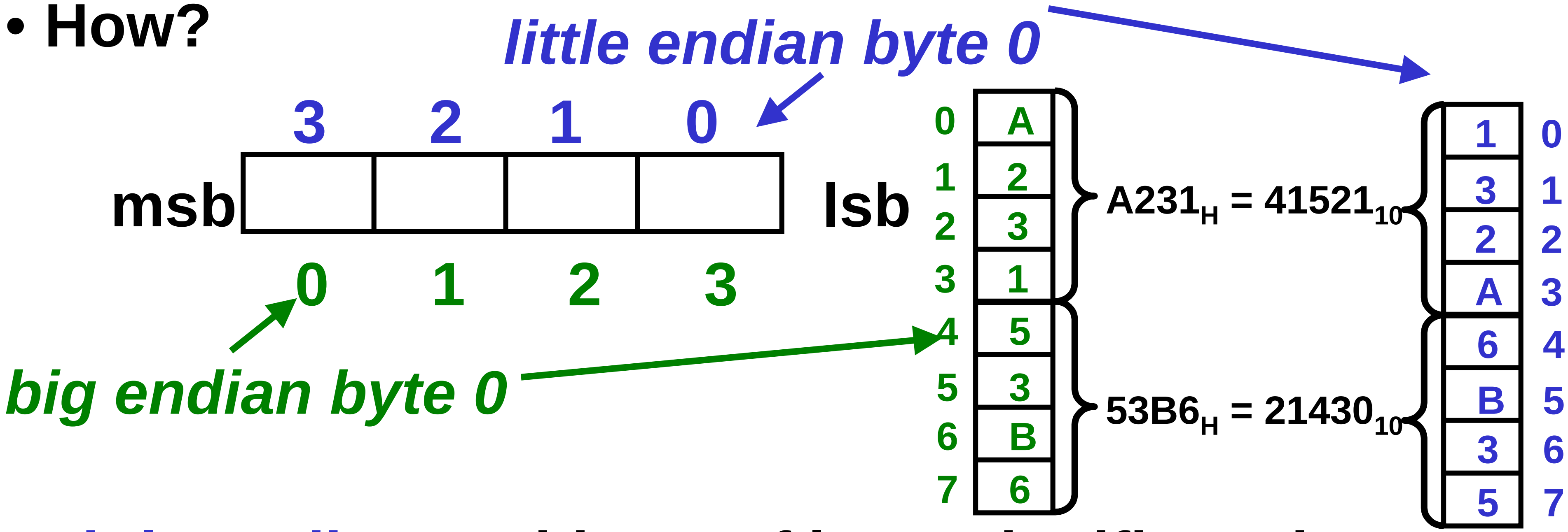
Memory: DRAM Architecture



http://web.mit.edu/rec/www/dramfaq/DRAMFAQ.html

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]]

Pseudo-Compilation using Memory

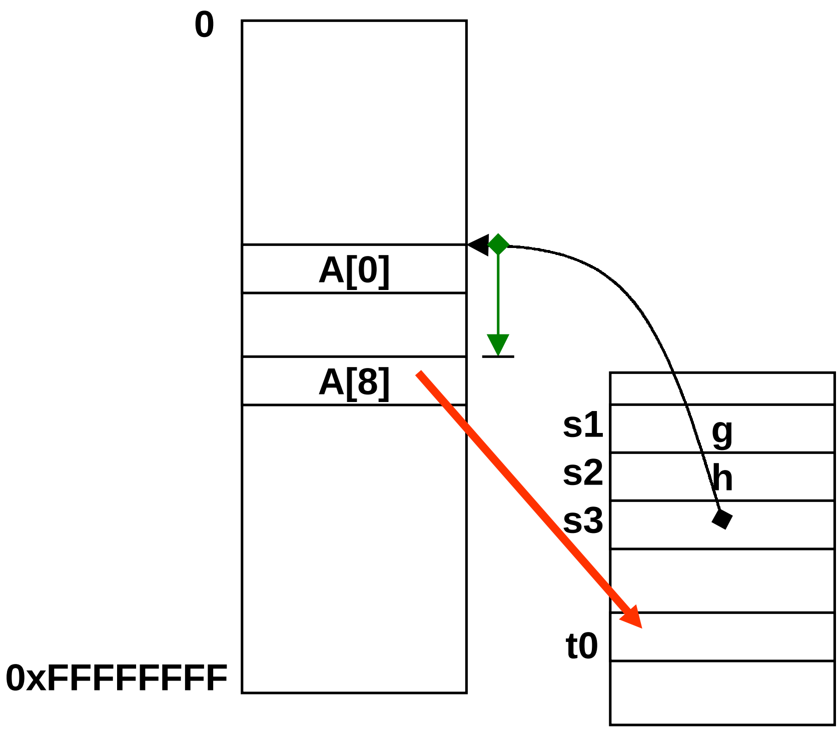
- Compile by hand (note: no register keyword):

  int f, g=5, h, i, j; f = (g + h) - (i + j);

- <u>MIPS Instructions:</u>

```
        lw   $s1,f($0)        # $s1 = mem[&f];
        lw   $s2,g($0)        # $s2 = mem[&g];
        lw   $s3,i($0)        # $s3 = mem[&i];
        lw   $s4,j($0)        # $s4 = mem[&j];
        add  $s0,$s1,$s2      # $s0 = g+h
        add  $t1,$s3,$s4      # $t1 = i+j
        sub  $s0,$s0,$t1      # f=(g+h)-(i+j)
        sw   $s0,f($0)        # mem[&f]=$s0
f       .word     0           #int f
g       .word     5           #int g ...
```

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 */
```

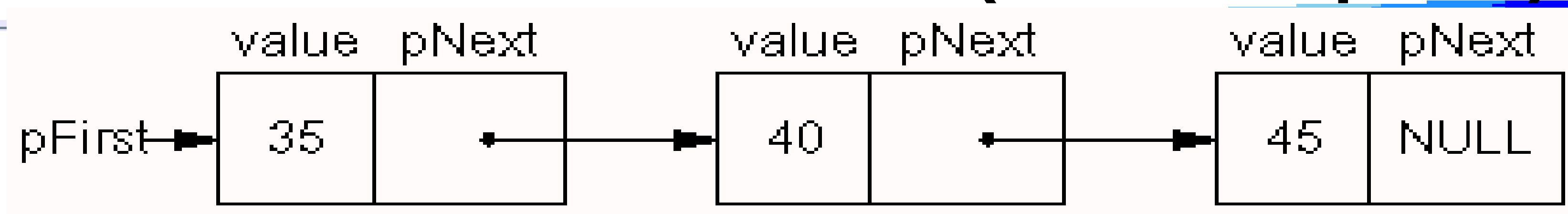
Compiled MIPS assembly instructions:

```
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

Pointers to structures (K & R chapter 5)

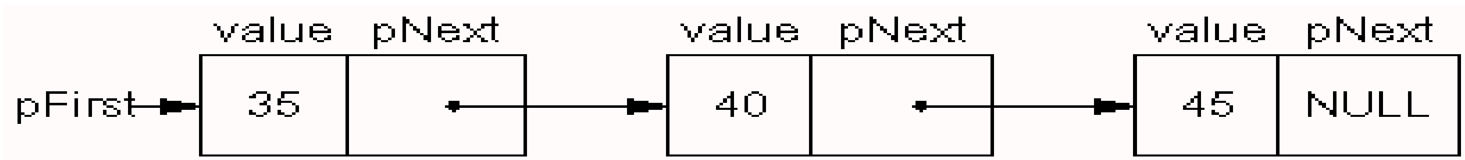



```
struct obj { int value; struct obj *pNext; }
struct obj object3 = { 45, NULL };
struct obj object2 = { 40, &object3 };
struct obj object1 = { 35, &object2 };
struct obj *pFirst = &object1;       /* pointer */
```

7 words of storage

```
         .data                  # struct obj { int value; struct obj *pNext; }
object3: .word      45          # struct obj object3 ={ 45, NULL };
         .word      0
object2: .word      40          # struct obj object2 ={ 40, &object3 };
         .word      object3
object1: .word      35          # struct obj object1 ={ 35, &object2 };
         .word      object2
pFirst:  .word      object1     # struct obj pFirst = &object1;
```

Static Memory Allocation (K & R chapter 5)




```
0x4000                 .data            # struct obj { int value; struct obj *pNext; }
0x4000    object3:  .word 45            # struct obj object3 ={ 45, NULL };
0x4004                 .word   0
0x4008    object2:  .word 40            # struct obj object2 ={ 40, &object3 };
0x400c                .word 0x4000      # &object3
0x4010    object1:  .word 35            # struct obj object1 ={ 35, &object2 };
0x4014                 .word 0x4008     # &object2 /* The address of object2 */
0x4018    pFirst:     .word 0x4010      # &object1   # struct obj pFirst = &object1;
```

Pointers to Structures: Modified example #1

```
struct obj { int value; struct obj *pNext; }
struct obj object3 = { 45, NULL };
struct obj object2 = { 40, object3 };
struct obj object1 = { 35, object2 };
struct obj *pFirst = object1;
```

Let's Change it

```
struct obj object3;
struct obj *object2;
struct obj **object1;
struct obj ***pFirst;
```

```
                .data
object3:        .word  0        # (int value) struct obj object3;
                .word  0        # (*pNext)
object2:        .word  0        # struct obj *object2;
object1:        .word  0        # struct obj **object1;
pFirst:         .word  0        # struct obj ***pFirst;
```

5 words of storage

Pointers to structures: Modified example #2

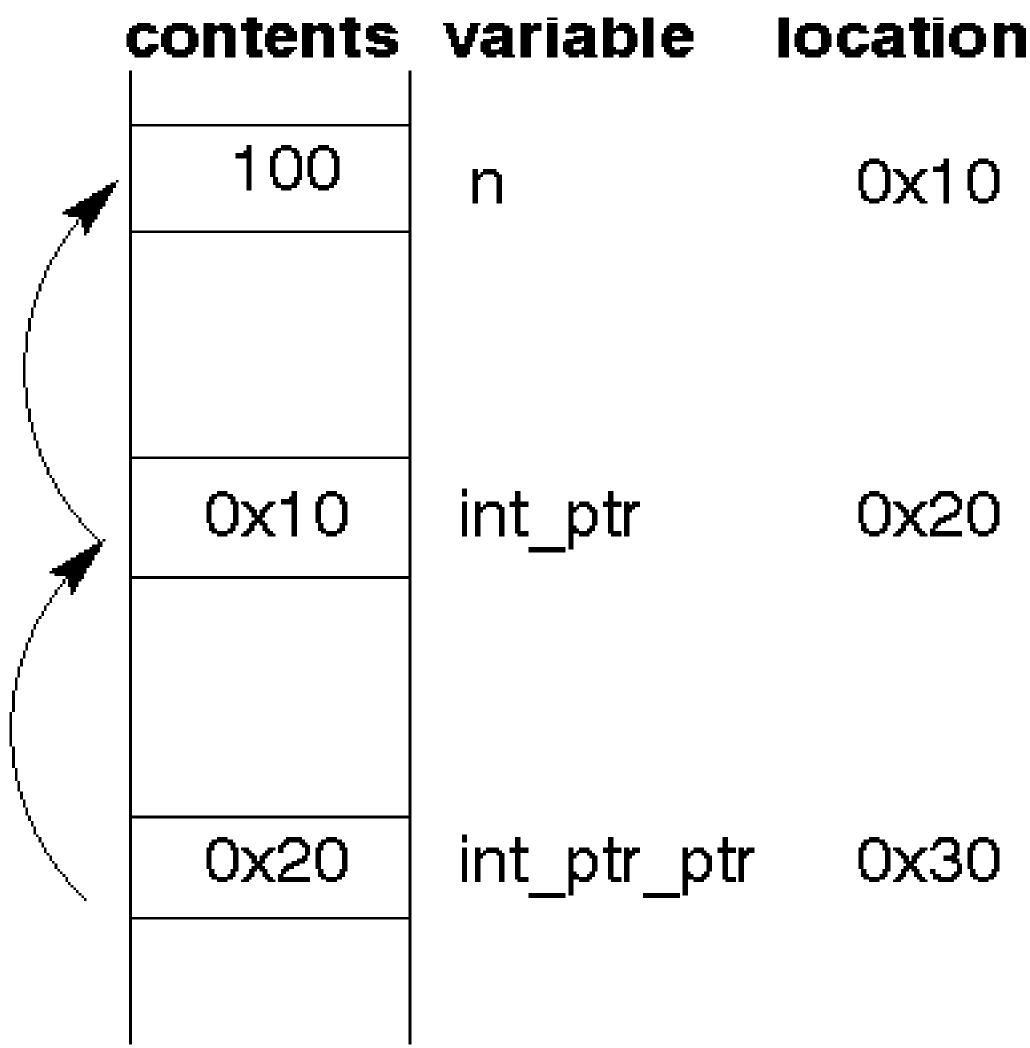
```
struct obj { int value; struct obj *pNext; }
struct obj object3   = { 35, NULL };
struct obj *object2  =  &object3;
struct obj **object1 = &object2;
struct obj ***pFirst  = &object1;
```

```
0x8000                  .data
0x8000  object3:        .word  35          # struct obj object3;
0x8004                  .word  0
0x800c  object2:        .word  0x8000      # struct obj *object2;
0x8010  object1:        .word  0x800c      # struct obj **object1;
0x8012  pFirst:         .word  0x8010      # struct obj *pFirst =&object1;
```

Pointers: multiple indirection (data)

```
int n;
int *int_ptr;
int **int_ptr_ptr;
int ***int_ptr_ptr;
int_ptr = &n;
int_ptr_ptr = &int_ptr;
**int_ptr_ptr = 100;
```

```
                .data
n:              .word  0
int_ptr:        .word  0
int_ptr_ptr:    .word  0
int_ptr_ptr_ptr: .word 0
```



Pointers: multiple indirection (code)

```
int n;
int *int_ptr;
int **int_ptr_ptr;
int ***int_ptr_ptr;
int_ptr = &n;
int_ptr_ptr = &int_ptr;
**int_ptr_ptr = 100;
```

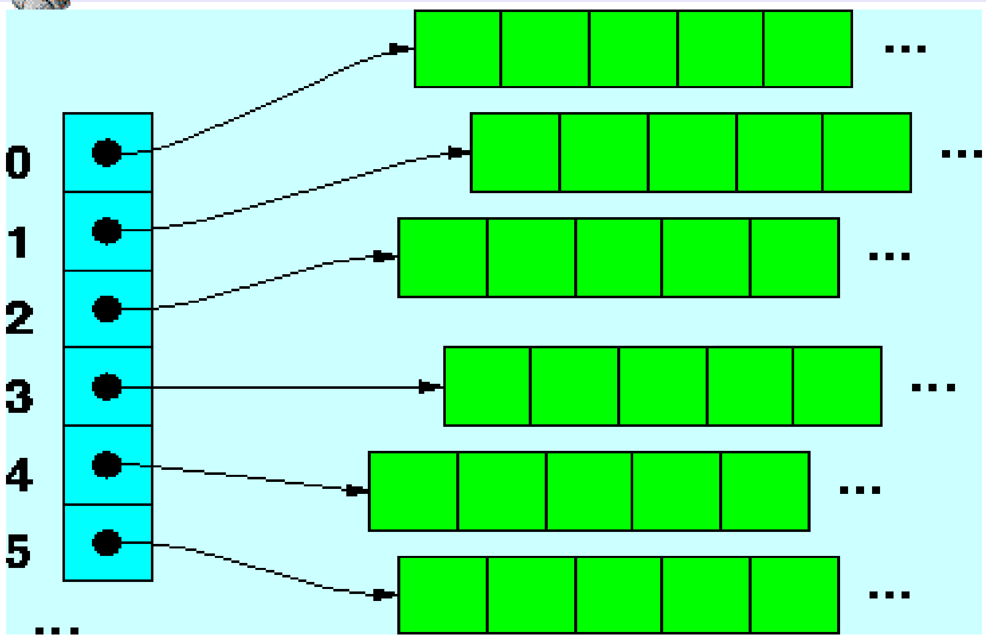
```
                .data
n:              .word  0
int_ptr:        .word  0
int_ptr_ptr:    .word  0
int_ptr_ptr_ptr: .word 0
```

```
            .text
la    $s0,n              # s0=&n;
la    $s1,int_ptr        # s1=&int_ptr;
sw    $s0,0($s1)         # *s1 = s0;

la    $s0,int_ptr        # s0 = &int_ptr
la    $s1,int_ptr_ptr    # s1= &int_ptr_ptr
sw    $s0,0($s1)         # *s0 = s1;

la    $s0,int_ptr_ptr    # s0=&int_ptr_ptr;
lw    $s1,0($s0)         # s1= *s0;
lw    $s2,0($s1)         # s2= *s1;
li    $s3,100            # s3 = 100;
sw    $s3,0($s2)         # *s3 = s2;
```

Pointers: An [ ]array of *pointers to int-egers



int ax[5], ay[5]; az[5];

int *argv[6] = { ax, ay, az };

```
        .data
ax:     .word  0,0,0,0,0
ay      .word  0,0,0,0,0
az      .word  0,0,0,0,0
argv:   .word  ax,ay,az,0,0,0
```

int *argv[6];

same as : int *(argv[6]);

Why is it 6 and not 5?

In C, the first index is 0 and last index is 5, which gives you a total of 6 elements.

```
        .data
argv:   .word  0,0,0,0,0,0
```

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, $0, or "$zero"

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; … */ .

3. See chalk board.