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.
Intel Corp. is the maker of semiconductor chips, supplies the computing and communications industries with chips, boards, systems and software that are integral in computers, servers, and networking and communications products. For the 39 weeks ended 9/29/01, revenues fell 22% to $19.56B. Net income fell 91% to $787M. Results reflect lower sales of microprocessors in the Intel Architecture Group and losses on equity investments. (Ref: money.cnn.com)

Traded: NASDAQ, INTC (http://money.langenberg.com/)
Sector: Technology
Industry: Semiconductors
Employees: 86,100
Market Cap (Mil)$: 240,222.48
Earnings / Share: 0.54
52-Week High / Low: 38.59 / 18.96
P/E: 64.41

Compagnie du jour: Intel Corporation www.intel.com
Review: Design Abstractions

- Coordination of many *levels of abstraction*

Performance Issues
- Speed
- Power
- Size

Software
- Application (Netscape)
- Operating System (Linux)
- Compiler
- Assembler

Hardware
- Processor
- Memory
- I/O system
- Datapath & Control
- Digital Design
- Circuit Design
- Transistors
Review: Design Abstractions

An abstraction omits unneeded detail, helps us cope with complexity.

temp = v[k];

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

v[k+1] = temp;

Compiler

Assembly Language Program (e.g. MIPS)

Assembler

Machine Language Program (MIPS)

Instruction Set Architecture

Control Signal Specification

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

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
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 */`

<table>
<thead>
<tr>
<th>Register</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r0$</td>
<td>0</td>
</tr>
<tr>
<td>$r1$</td>
<td>1</td>
</tr>
<tr>
<td>$r2$</td>
<td>2</td>
</tr>
<tr>
<td>$r3$</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$r28$</td>
<td>28</td>
</tr>
<tr>
<td>$r29$</td>
<td>29</td>
</tr>
<tr>
<td>$r30$</td>
<td>30</td>
</tr>
<tr>
<td>$r31$</td>
<td>31</td>
</tr>
</tbody>
</table>
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
## MIPS registers and conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Conventional usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Expression evaluation &amp; function results</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments 1 to 4</td>
</tr>
<tr>
<td>$t1-$t9</td>
<td>8-15,24,35</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved Temporary (preserved across call)</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Pointer to global area</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address (used by function call)</td>
</tr>
</tbody>
</table>
Calculate (PH p. 109, file: simplecalc.s)

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

---

### MIPS Register Name translation

```c
addi $s1, $0, 5  \quad \rightarrow \quad addi $17, $0, 5 \quad # g = 5
addi $s2, $0, -20 \quad \rightarrow \quad addi $18, $0, -20 \quad # h = -20
addi $s3, $0, 13 \quad \rightarrow \quad addi $19, $0, -20 \quad # i = 13
addi $s4, $0, 3 \quad \rightarrow \quad addi $20, $0, 3 \quad # j = 3
add $t0, $s1, $s2 \quad \rightarrow \quad add $8, $17, $18 \quad # $t0=g + h
add $t1, $s3, $s4 \quad \rightarrow \quad add $9, $19, $20 \quad #$t1=i + j
sub $s0, $t0, $t1 \quad \rightarrow \quad sub $16, $8, $9 \quad # f=(g+h)-(i+j)
```
SPIM: System call 1: print_int $a0

• System calls are used to interface with the operating system to provide device independent 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)`

Assembler .s

Translated (1 to 1 mapping)

```
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

<table>
<thead>
<tr>
<th>Service</th>
<th>Code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0=integer</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12=float</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12=double</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0=string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td>$v0=integer</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td>$f0=float</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td></td>
<td>$f0=double</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0=buf, $a1=len</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0=amount</td>
<td>$v0=address</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)`

**Assembler .s**

```
.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
```

**Translated (1 to 1 mapping)**

```
msg3 is just a label but must match
ori $2, $0, 4    #print_string
lui $4,4097     #address of string
syscall syscall
```
.asciiz data representations

.data: items are placed in the data segment
    which is not the same as the .text segment!

Assemler .s

msg3: .asciiz "\nThe va"

Same as in assembler.s


Same as in assembler.s

msg3: .byte 0x0a, 0x54, 0x68, 0x65
    .byte 0x20, 0x76, 0x61, 0x00

Same as in assembler.s

msg3: .word 0x6568540a, 0x00617620

Translated in the .data segment: 0x6568540a 0x00617620
Segments allow the operating system to protect memory
Like Unix file systems: .text Execute only, .data R/W only

- .text segment
- .data segment
- .stack segment

Memory layout: segments

```assembly
addi $17,0,5
addi $18,0,-20
```

.$sp = top of stack

```assembly
.ascii "The value of f is "
```

0x00400000

0x7fffffff
Hello, World: hello.s

# 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

Note: alternating .text, .data, .text
Simplecalc.s (PH p. 109)

.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

Order of .text and .data not important
## Simplecalc.s without symbols (PH p. 109)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Values</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400020</td>
<td>addu</td>
<td>$23, $0, $31</td>
<td># addu $s7, $0, $ra</td>
</tr>
<tr>
<td>0x00400024</td>
<td>addi</td>
<td>$17, $0, 5</td>
<td># addi $s1, $0, 5</td>
</tr>
<tr>
<td>0x00400028</td>
<td>addi</td>
<td>$18, $0, -20</td>
<td># addi $s2, $0, -20</td>
</tr>
<tr>
<td>0x0040002c</td>
<td>addi</td>
<td>$19, $0, 13</td>
<td># addi $s3, $0, 13</td>
</tr>
<tr>
<td>0x00400030</td>
<td>addi</td>
<td>$20, $0, 3</td>
<td># addi $s4, $0, 3</td>
</tr>
<tr>
<td>0x00400034</td>
<td>add</td>
<td>$8, $17, $18</td>
<td># add $t0, $s1, $s2</td>
</tr>
<tr>
<td>0x00400038</td>
<td>add</td>
<td>$9, $19, $20</td>
<td># add $t1, $s3, $s4</td>
</tr>
<tr>
<td>0x0040003c</td>
<td>sub</td>
<td>$16, $8, $9</td>
<td># sub $s0, $t0, $t1</td>
</tr>
<tr>
<td>0x00400040</td>
<td>ori</td>
<td>$2, 0, 4</td>
<td>#print_str (system call 4)</td>
</tr>
<tr>
<td>0x00400044</td>
<td>lui</td>
<td>$4, 0x10010000</td>
<td># address of string</td>
</tr>
<tr>
<td>0x00400048</td>
<td>syscall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0040004c</td>
<td>ori</td>
<td>$2, 1</td>
<td>#print_int (system call 1)</td>
</tr>
<tr>
<td>0x00400050</td>
<td>add</td>
<td>$4, $0, $16</td>
<td>#put value to print in $a0</td>
</tr>
<tr>
<td>0x00400054</td>
<td>syscall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00400058</td>
<td>addu</td>
<td>$31, $0, $23</td>
<td>#restore the return address</td>
</tr>
<tr>
<td>0x0040005c</td>
<td>jr</td>
<td>$31</td>
<td>#return to the main program</td>
</tr>
<tr>
<td>0x00400060</td>
<td>add</td>
<td>$0, $0, $0</td>
<td>#nop</td>
</tr>
</tbody>
</table>

### .data

<table>
<thead>
<tr>
<th>Address</th>
<th>.word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010000</td>
<td>0x6568540a, 0x6c617620, 0x6f206575</td>
</tr>
<tr>
<td></td>
<td>0x20662066, 0x203a7369, 0x00000000</td>
</tr>
<tr>
<td>$pc</td>
<td>$t0</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>00400030</td>
<td>?</td>
</tr>
<tr>
<td>00400034</td>
<td>?</td>
</tr>
<tr>
<td>00400038</td>
<td>fffffff1</td>
</tr>
<tr>
<td>0040003c</td>
<td>?</td>
</tr>
<tr>
<td>00400040</td>
<td>?</td>
</tr>
</tbody>
</table>
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);
  ```

- 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)
  ```

Note: whereas C declares its operands, Assembly operands (registers) are fixed and not declared.
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:
  
  ```c
  register int f, g, h, i, j;
  f = (g + h) - (i + j);
  ```

- 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.

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

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

<table>
<thead>
<tr>
<th>0</th>
<th>8 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>2</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>3</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>5</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>6</td>
<td>8 bits of data</td>
</tr>
</tbody>
</table>

...
Memory Organization: words

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

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

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

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

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

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

```
0     1     2     3
Aligned
Not Aligned
```

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

- (Later we’ll see how alignment helps performance)
Memory: Bus

- Direct AGP
- AGP
- EIDE
- AC97 (audio, modem)
- Memory Controller Hub
- I/O Controller Hub
- CPU
- Interlink Bus
- RAM
- PCI bus
- USB

Internal Architecture of a DRAM
Memory: Dram 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 byte 0**

  ![Diagram of little endian byte 0]

  **big endian byte 0**

- 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**: \( \text{lw} \ $t0, \ 8($s3) \)

  Called “offset”  Called “base register”

- **MIPS lw semantics**: \( \text{reg}[t0] = \text{Memory}[8 + \text{reg}[s3]] \)
Pseudo-Compilation using Memory

- Compile by hand (note: no register keyword):
  
  ```c
  int f, g=5, h, i, j; f = (g + h) - (i + j);
  ```

- MIPS Instructions:

  ```
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 ...
```
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

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

```c
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: \( g = h + A[i] \);

<table>
<thead>
<tr>
<th>C Variables</th>
<th>( g )</th>
<th>( h )</th>
<th>( A )</th>
<th>( i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>$s1 $s2</td>
<td>$s3 $s4</td>
<td>$t0 $t1</td>
<td></td>
</tr>
</tbody>
</table>

- **suppose** \( \text{mem}[3020] = 38 \)
  - \$t0, \$t1: 4, 3000
  - \$s1, \$s2, \$s3, \$s4: 38, 20

- **Add** \$t1, \$s4, \$s4
  - \$t0, \$t1: 4, 3000

- **Add** \$t1, \$t1, \$t1
  - \$t0, \$t1: 4, 3000

- **Add** \$t1, \$t1, \$s3
  - \$t0, \$t1: 4, 3000

- **Lw** \$t0, 0(\$t1)
  - \$t0, \$t1: 4, 3000

- **Add** \$s1, \$s2, \$t0
  - \$s1, \$s2, \$t0: 38, 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 */
```

```
data
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

```c
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;
```

```
.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;
```
Pointers to structures: Modified example #2

```c
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)

```c
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;
```

<table>
<thead>
<tr>
<th>contents</th>
<th>variable</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>n</td>
<td>0x10</td>
</tr>
<tr>
<td>0x10</td>
<td>int_ptr</td>
<td>0x20</td>
</tr>
<tr>
<td>0x20</td>
<td>int_ptr_ptr</td>
<td>0x30</td>
</tr>
</tbody>
</table>

```
.data
n: .word 0
int_ptr: .word 0
int_ptr_ptr: .word 0
int_ptr_ptr_ptr: .word 0
```
Pointers: multiple indirection (code)

```c
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;
```

CWRU EECS 314  40
Pointers: An [ ] array of *pointers to integers

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.

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

int *argv[6] = { ax, ay, az }; .data
argv: .word ax,ay,az,0,0,0
```

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

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

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

int *argv[6] = { ax, ay, az }; .data
argv: .word ax,ay,az,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

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

i = i + limit;
```

Is the same as

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

And the compiler will protect you from doing this

```c
limit = 5;
```
Class Homework: Due next class

C code fragment:

```c
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:
   
   ```c
   A=address 1000, B=address 5000, i=3, h=10, k=2, 
   /* i.e. A[0]=24; A[1]=33; … */.
   ```

3. See chalk board.