Review 1/3

- MIPS assembly language instructions mapped to numbers in 3 formats

<table>
<thead>
<tr>
<th>R</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>op</td>
<td>address</td>
<td></td>
</tr>
</tbody>
</table>

- Op field determines format

- Operands
  - Registers: $0$ to $31$ mapped onto $\$zero$; $\$at$; $\$v0$, $\$v1$, $\$a0.., \$s0.., \$t0.., \$gp$, $\$sp$, $\$fp$, $\$ra$
  - Memory: $\text{Memory}[0], \text{Memory}[4], \text{Memory}[8], ..., \text{Memory}[4294967292]
    - Index is the address of the word

Instructions/Formats/“opcodes”, Regs

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Format</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Register</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>sub</td>
<td>Register</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>slt</td>
<td>Register</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>jr</td>
<td>Register</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>lw</td>
<td>Immediate</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>Immediate</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>Immediate</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>Immediate</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>Immediate</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>slti</td>
<td>Immediate</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>Jump</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td>Jump</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- Name | Number
- $\$zero$ | $0$
- $\$at$ | $1$
- $\$v0$, $\$v1$ | $2$-$3$
- $\$a0.., \$s0.., \$t0..$ | $4$-$7$
- $\$gp$, $\$sp$, $\$fp$, $\$ra$ | $26$, $27$
- $\$t8-$t9$ | $23$-$24$
- $\$k0, \$k1$ | $26$, $27$
- $\$gp$ | $28$
- $\$sp$ | $29$
- $\$fp($t10)$ | $30$
- $\$ra$ | $31$
Overview
° Decoding Instructions from numbers
° Why MIPS?
° C Function Memory Allocation review
° Administrivia, “Cool Technology”
° 4 Versions of C/Asm. sumarray function
° Pointers and memory allocation in C
° Conclusion

Decoding example
° Binary=>Decimal=>Assembly=>C?
° Start at program at address 4,194,304 (2^{22})

```
00000000000000000001000000100000
000000000000010101001000001010100
000000000000010010010000001010100
00100000101010011111111111111111
00000000000001010100100000101010
0001010100100000111111111111111
```
° What are instruction formats of these 7 instructions?

Decoding example: Binary=>Decimal
° Binary=>Decimal=>Assembly=>C?

```
R
RIRIR
I
00203 2
005904 2
024203 2
005904 2
85 5 - 1
49 0 5
59 0 - 4
```

Decoding example: Decimal=>Assembly
° Decimal=>Assembly? (Slide 4)

```
4194304 add $2, $0, $0
4194308 slt $9, $0, $5
4194312 beq $9, $0, 5
4194316 add $2, $2, $4
4194320 addi $5, $5, $1
4194324 slt $9, $0, $5
4194328 bne $9, $0, $4
```

Decoding example: Binary=>Decimal
° Binary=>Decimal=>Assembly=>C?
### Decoding example: Symbolic Assembly

**Registers, see Slide 4**

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>4194304</td>
<td>add</td>
<td>$2, $0, $0</td>
</tr>
<tr>
<td>4194308</td>
<td>slt</td>
<td>$9, $0, $5</td>
</tr>
<tr>
<td>4194312</td>
<td>beq</td>
<td>$9, $0, 5</td>
</tr>
<tr>
<td>4194316</td>
<td>add</td>
<td>$2, $2, $4</td>
</tr>
<tr>
<td>4194320</td>
<td>addi</td>
<td>$5, $5, -1</td>
</tr>
<tr>
<td>4194324</td>
<td>slt</td>
<td>$9, $0, $5</td>
</tr>
<tr>
<td>4194328</td>
<td>bne</td>
<td>$9, $0, -4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4194324</td>
<td>slt $t1,$zero,$a1</td>
</tr>
<tr>
<td>4194328</td>
<td>bne $t1,$zero,Loop</td>
</tr>
</tbody>
</table>

### Mapping

```
product = 0;
while (0 < mlier) {
    product = product + mcand;
    mlier = mlier - 1;
}
```

### Why MIPS?

- **Example of modern design: RISC, or Reduced Instruction Set Computer**
  - Style of computer, e.g., sports car
  - RISC developed at IBM, Berkeley, Stanford

- **RISCs much easier to understand v. x86**

- **1998 32-bit Embedded Processors**
  - 83M: Motorola 680x0
  - 50M: MIPS
  - 48M: ARM (Acorn RISC Machine)
  - 26M: Hitachi SuperH (RISC)
  - 12M: x86 (5% of market)

### Review: C memory allocation

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
<th>Space for saved procedure information</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td>Explicitly created space, e.g., malloc(); C pointers</td>
</tr>
<tr>
<td>$gp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>global pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td></td>
<td>Variables declared once per program</td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td>Program</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Review: Memory Allocation on Call

C Procedure Call Frame

- Pass arguments (4 regs)
- Save caller-saved regs
- jal
- space on stack ($sp-n)
- save $fp & set $fp 1st word of frame ($sp+n-4)
- Save callee-saved regs
- If may call functions, save arguments and $ra

Saved Registers

<table>
<thead>
<tr>
<th>Argument 6</th>
<th>Argument 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fp</td>
<td>Saved</td>
</tr>
<tr>
<td></td>
<td>Registers</td>
</tr>
<tr>
<td></td>
<td>Local</td>
</tr>
<tr>
<td></td>
<td>Variables</td>
</tr>
</tbody>
</table>

Address
grows

Stack Area

low

high

Argument Passing Options

- 2 choices
  - “Call by Value”: pass a copy of the item to the function/procedure
  - “Call by Reference”: pass a pointer to the item to the function/procedure

- Single word variables passed by value
- Passing an array? e.g., a[100]
  - Pascal--call by value--copies 100 words of a[] onto the stack
  - C--call by reference--passes a pointer (1 word) to the array a[] in a register

Review: Memory Deallocation on Return

- Move return value into $v0
- Restore callee-saved regs from the stack
- If saved $ra, restore it
- Restore $fp from stack
- Pop stack ($sp+n)
- jr $ra
- Restore caller-saved regs

Address
grows

(low)

Administriivia

- Readings: 3.7, 4.1, 4.2, 4.3
- 3rd homework: Due Wed 2/10 7PM
  - Exercises 3.7, 3.8, 3.10
- 2nd project: MIPS Disassembler
  Due Wed. 2/17 7PM
- Midterm, Final: 5-8PM (3/17, 5/12)
  - Conflicts: email mds@cory
Cool Technology: DIMM-PC/486

- 486 PC smaller than a credit card

- ELAN410 66MHz (486) CPU, 16 MB of DRAM, 16 MB of flash; all standard PC interfaces, such as 2 serial ports, printer, floppy and a hard disk interface; Ethernet option, Redhat 5.2 Linix: www.jumptec.com

- 1.6” x 2.7” x 0.25”, 0.8 w standby, $419,

“Computers in the News”

- “Intel Alters Plan Said To Undermine PC Users' Privacy”, 1st p., N.Y. Times 1/26/99
- Processor-specific IDs per chip accessed by SW and transmitted over the Internet
  - 96-bit unique serial number: 32 CPU type+64 ID
  - Idea: ID helps intellectual property protection, tying apps, information to a specific machine
- “Big Brother” inside? Boycott Intel!
  - No anonymity? Track 1 consumer over Internet?

- “The Intel Corporation yesterday reversed a plan to activate an identifying signature in its next generation of computer chips, bowing to protests that the technology would compromise the privacy of users.”
  - Not removed; default now off on reboot

Arrays, Pointers, Functions in C

- 4 versions of array function that adds two arrays and puts sum in a third array (sumarray)
  - Third array is passed to function
  - Using a local array (on stack) for result and passing a pointer to it
  - Third array is allocated on heap
  - Third array is declared static

- Purpose of example is to show interaction of C statements, pointers, and memory allocation

Calling sumarray, Version 1

```c
int x[100], y[100], z[100];
sumarray(x, y, z);
```

- C calling convention means above the same as
  ```c
  sumarray(&x[0], &y[0], &z[0]);
  ```

- Really passing pointers to arrays
  ```c
  addi $a0,$gp,0      # x[0] starts at $gp
  addi $a1,$gp,400    # y[0] above x[100]
  addi $a2,$gp,800    # z[0] above y[100]
  jal sumarray
  ```
**Version 1: Optimized Compiled Code**

```c
void sumarray(int a[], int b[], int c[]) {
    int i;
    for (i = 0; i < 100; i = i + 1)
        c[i] = a[i] + b[i];
}
```

```assembly
addi $t0, $a0, 400 # beyond end of a[]
Loop: beq $a0, $t0, Exit
lw $t1, 0($a0) # $t1 = a[i]
lw $t2, 0($a1) # $t2 = b[i]
add $t1, $t1, $t2 # $t1 = a[i] + b[i]
sw $t1, 0($a2) # c[i] = a[i] + b[i]
addi $a0, $a0, 4 # $a0++
addi $a1, $a1, 4 # $a1++
addi $a2, $a2, 4 # $a2++
j Loop
Exit: jr $ra
```

**Version 1: Before optimizing compilers**

```c
void sumarray(int a[], int b[], int c[]) {
    int i, c[100];
    for (i = 0; i < 100; i = i + 1)
        c[i] = a[i] + b[i];
}
```

```assembly
addi $t0, $a0, 400 # beyond end of a[]
Loop: beq $a0, $t0, Exit
lw $t1, 0($a0) # $t1 = a[i]
lw $t2, 0($a1) # $t2 = b[i]
add $t1, $t1, $t2 # $t1 = a[i] + b[i]
sw $t1, 0($a2) # c[i] = a[i] + b[i]
addi $a0, $a0, 4 # $a0++
addi $a1, $a1, 4 # $a1++
addi $a2, $a2, 4 # $a2++
j Loop
Exit: jr $ra
```

**Version 2 to Fix Weakness of Version 1**

- **Would like recursion to work**

```c
int sumarray(int a[], int b[]);
/* adds 2 arrays and returns sum */
sumarray(x, sumarray(y, z));
```

- **Cannot do this with Version 1 style solution: what about this**

```c
int * sumarray(int a[], int b[]) {
    int i, c[100];
    for (i = 0; i < 100; i = i + 1)
        c[i] = a[i] + b[i];
    return c;
}
```

**Version 2: Revised Compiled Code**

```c
for (i = 0; i < 100; i = i + 1)
    c[i] = a[i] + b[i];
return c;
```

```assembly
addi $t0, $a0, 400 # beyond end of a[]
addi $sp, $sp, -400 # space for c
addi $t3, $sp, 0 # ptr for c
addi $v0, $t3, 0 # $v0 = &c[0]
Loop: beq $a0, $t0, Exit
lw $t1, 0($a0) # $t1 = a[i]
lw $t2, 0($a1) # $t2 = b[i]
add $t1, $t1, $t2 # $t1 = a[i] + b[i]
sw $t1, 0($t3) # c[i] = a[i] + b[i]
addi $a0, $a0, 4 # $a0++
addi $a1, $a1, 4 # $a1++
addi $a2, $a2, 4 # $a2++
j Loop
Exit: addi $sp, $sp, 400 # pop stack
jr $ra
**Weakness of Version 2**

- Legal Syntax; What’s Wrong?
- Will work until call another function that uses stack
- Won’t be reused instantly (e.g., add a printf)
- Stack allocated + unrestricted pointer is problem

**Version 3 to Fix Weakness of Version 2**

- Solution: allocate c[] on heap

```c
int * sumarray(int a[], int b[])
{
    int i;
    int *c;
    c = (int*) malloc(100);
    for (i=0; i<100; i=i+1)
        c[i] = a[i] + b[i];
    return c;
}
```

- Not reused unless freed
  - Can lead to memory leaks
  - Java, Scheme have garbage collectors to reclaim free space

**Version 3: Revised Compiled Code**

```c
addi $t0, $a0, 400 # beyond end of a[]
addi $sp, $sp, -12 # space for regs
sw $ra, 0($sp) # save $ra
sw $a0, 4($sp) # save 1st arg.
sw $a1, 8($sp) # save 2nd arg.
addi $a0, $zero, 400 #
jal malloc
addi $t3, $v0, 0 # ptr for c
lw $a0, 4($sp) # restore 1st arg.
lw $a1, 8($sp) # restore 2nd arg.
```

**Version 4: Alternative to Version 3**

- Static declaration

```c
int * sumarray(int a[], int b[])
{
    int i;
    static int c[100];
    for (i=0; i<100; i=i+1)
        c[i] = a[i] + b[i];
    return c;
}
```

- Compiler allocates once for function, space is reused
  - Will be changed next time sumarray invoked
  - Why describe? used in C libraries
What about Structures?
° Scalars passed by value
° Arrays passed by reference (pointers)
° Structures by value too
° Can think of C passing everything by value, just that arrays are simply a notation for pointers and the pointer is passed by value

“And in Conclusion …” 1/2
° MIPS assembly language instructions mapped to numbers in 3 formats

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address</td>
</tr>
</tbody>
</table>
° Op field determines format
° Binary => Decimal => Assembly => Symbolic Assembly => C
• Reverse Engineering or Disassembly
• Its hard to do, therefore people like shipping binary machine language more than assembly or C

“And in Conclusion …” 2/2
° Programming language model of memory allocation and pointers
  • Allocate in stack vs. heap vs. global areas
  • Arguments passed
call by value vs. call by reference
  • Pointer in C is HLL version of machine address
° Next : character, strings, other numbers