MIPS Architecture

• Topics
  – What resources MIPS assembly manipulate
  – Representing information as bits
    • Binary/Hexadecimal
    • Byte representations
      – numbers
      – characters and strings
      – Instructions
MIPS MARS simulator

• Please download, install, and get familiar with
• http://courses.missouristate.edu/KenVollmar/MARS/download.htm
MIPS Architecture
Assembly Programmer’s View

Memory
- Object Code
- Program Data
- OS Data

Stack

CPU
- Registers
- Condition Codes

• Programmer-Visible State
  - Program Counter (PC)
    - Address of next instruction
  - Location Counter (LOC)
  - Register File
    - Heavily used program data
  - Condition Codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

Memory
- Byte addressable array
- Code, user data, (some) OS data
- Includes stack

Addresses
Data
Instructions
# Registers

<table>
<thead>
<tr>
<th>Reg. #</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
<td>the value 0</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>(assembler temporary) reserved by the assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0 - $v1</td>
<td>(values) from expression evaluation and function results</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0 - $a3</td>
<td>(arguments) First four parameters for subroutine. Not preserved across procedure calls</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0 - $t7</td>
<td>(temporaries) Caller saved if needed. Subroutines can use w/ out saving. Not preserved across procedure calls</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0 - $s7</td>
<td>(saved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls</td>
</tr>
<tr>
<td>24-25</td>
<td>$t8 - $t9</td>
<td>(temporaries) Caller saved if needed. Subroutines can use w/ out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.</td>
</tr>
<tr>
<td>26-27</td>
<td>$k0 - $k1</td>
<td>reserved for use by the interrupt/trap handler</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>global pointer. Points to the middle of the 64K block of memory in the static data segment.</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>stack pointer Points to last location on the stack.</td>
</tr>
<tr>
<td>30</td>
<td>$s8/$fp</td>
<td>saved value / frame pointer Preserved across procedure calls</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>return address</td>
</tr>
</tbody>
</table>
Registers

- 32-bit 32 Registers
  - Zero: always 0, and cannot be modified
  - $t0 -- $t9: General purpose
  - $a0 -- $a3: General purpose (arguments)
  - $s0 -- $s7: General purpose
  - $sp: stack pointer
  - $ra: return address
Moving Data

• Moving Data
  
  \texttt{lw Dest, Source:}
  
  – Move 4-byte (“long”) word
  – Constant or from memory address
  – To Dest register

• Operand Types
  
  – Immediate: Constant integer data
    
    • 0xff for hex constant
    • Otherwise, decimal
  – Memory: 4 consecutive bytes of memory
    
    • Various “address modes”
  – Register: One of 32 integer registers
## Operand Addressing

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>li $t1, 0x4</td>
</tr>
<tr>
<td>Addr</td>
<td>Reg</td>
<td>la $t1, A</td>
</tr>
<tr>
<td>Reg</td>
<td>Mem</td>
<td>sw $t1,A($t2)</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>lw $t1,A($t2)</td>
</tr>
</tbody>
</table>

- No instruction for reg-to-reg transfer
- Cannot do memory-memory transfers with single instruction
- sw instruction violates (dest, source) spec of operands
Memory Accesses

• Addressing Modes

  • Indirect  \( (R) \)  \( \text{Mem}[	ext{Reg}[R]] \)
    – Register R specifies memory address
      \[ \text{lw} \ t1, (t2) \]

  • Indexed  \( D(R) \)  \( \text{Mem}[	ext{Reg}[R]+D] \)
    – Register R specifies start of memory block
    – Constant displacement D specifies offset
      \[ \text{lw} \ t1, 8(t2) \]
Example

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
# xp in $a0, yp in $a1
swap:
    lw $t0,($a0)    # t0=*xp
    lw $t1,($a1)    # t1=*yp
    sw $t1,($a0)    # *xp=t1
    sw $t0,($a1)    # *yp=t0
    jr $ra
```
Understanding Swap

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

# xp in $a0, yp in $a1
swap:
    lw $t0, ($a0)
    lw $t1, ($a1)
    sw $t1, ($a0)
    sw $t0, ($a1)
    jr $ra
# xp in $a0, yp in $a1

swap:

lw  $t0, ($a0)
lw  $t1, ($a1)
sw  $t1, ($a0)
sw  $t0, ($a1)
jr  $ra
# xp in $a0, yp in $a1

swap:

```
lw $t0, ($a0)
lw $t1, ($a1)
sw $t1, ($a0)
sw $t0, ($a1)
jr $ra
```
# xp in $a0, yp in $a1

swap:

lw    $t0, ($a0)
lw    $t1, ($a1)
sw    $t1, ($a0)
sw    $t0, ($a1)
jr    $ra

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>xp</td>
<td>120</td>
</tr>
<tr>
<td>$a1</td>
<td>yp</td>
<td>124</td>
</tr>
<tr>
<td>$t0</td>
<td></td>
<td>456</td>
</tr>
<tr>
<td>$t1</td>
<td></td>
<td>123</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x124</td>
<td>123</td>
</tr>
</tbody>
</table>
# xp in $a0, yp in $a1

swap:

lw  $t0, ($a0)
lw  $t1, ($a1)
sw  $t1, ($a0)
sw  $t0, ($a1)
jr  $ra

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
<th>Value</th>
<th>Value (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>xp</td>
<td>120</td>
<td>0x124</td>
</tr>
<tr>
<td>$a1</td>
<td>yp</td>
<td>124</td>
<td>0x120</td>
</tr>
<tr>
<td>$t0</td>
<td></td>
<td>456</td>
<td>456</td>
</tr>
<tr>
<td>$t1</td>
<td></td>
<td>123</td>
<td>123</td>
</tr>
</tbody>
</table>
MIPS resources

• Instruction Reference
  http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

• Architecture and Assembly Language Overview
  http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm
Arithmetic Operations

- **add** $t0, t1, t2
  - $t0 = $t1 + $t2; add as signed (2's complement) integers
  - (addi $t1, $t2, 0x32)
- **sub** $t1, t2, t2
- **mul** $t1, t2, t2
- **div** $t1, t2, t2
- **and** (andi), or (ori), xor (xori)
- **sll** $t1, t2, 2 (shift left logical)
- **sllv** (sll variable)
- **sra** (shift right arithmetic)
- **srl**
Branch Instructions

• b (unconditional)

• Conditional
  – beq, bne
  – bgez, bgtz, blez, bltz
  – bgezal bltzal

• Set Condition
  – Sltz
## I/O Instructions

<table>
<thead>
<tr>
<th>Service</th>
<th>Call code ($v0)</th>
<th>Arguments (input)</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>$a0 = integer</td>
<td>signed decimal integer printed in console window</td>
</tr>
<tr>
<td>print string</td>
<td>4</td>
<td>$a0 = address of string</td>
<td>string printed in console window</td>
</tr>
<tr>
<td>Read integer</td>
<td>5</td>
<td>(none)</td>
<td>$v0 holds integer that was entered</td>
</tr>
<tr>
<td>Read string</td>
<td>8</td>
<td>$a0=address to store $a1= length limit</td>
<td>characters are stored</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td>(none)</td>
<td>Ends the program</td>
</tr>
</tbody>
</table>
Example: Hello World

.data
st_hello: .asciiz "Hello, world ! \n" # message area

.text
.globl main
main:
    la $a0, st_hello # get the address of message to $a0
    li $v0, 4      # read to display the message
    syscall
    li $v0, 10     # quit the program
    syscall
Directives

- Directives (Establish initial data structure)
  - .ascii (store string in memory with no null-termination)
  - .asciiz (store string in memory with null termination)
  - .byte b1,..,bn
  - .word w1,..,wn
  - .word w:n (store w into n successive words)
  - .space n

- .data data segment
- .text assembly instructions
Resources

• MIPS instructions
  • [http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html](http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html)

• MIPS Reading

• MARS MIPS simulator
  • [http://courses.missouristate.edu/KenVollmar/MARS/](http://courses.missouristate.edu/KenVollmar/MARS/)
Example: Square an input Number

```
printf(“Enter an integer to square: \n”);
scanf(“%d”, x);
printf(“%4d”, (x*x));
```

```
.data
prompt: .asciiz “Enter an integer to square:\n“ # message area

.text
.globl main
main: # printf(“Enter an integer to square: \n”);
la $a0, prompt # get the address of the message to $a0
li $v0, 4 # read to display the message
syscall

# scanf(“%d”, x);
li $v0, 5 # read an integer into $a0
syscall

mul $a0, $v0, $v0 # squared input value, save in $a0

# printf(“%4d”, (x*x));
li $v0, 1 # print the squared value
syscall
```

Example:
Square an input Number

```
23!
```
int max(int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}

# x in $a0, y in $a1
# return value in $t0
max:
    bgt  $a0, $a1, ret_x
    or $t0, $a1, $0
    jr   $ra

ret_x:
    or $t0, $a0, $0
    jr   $ra
Example: Add N-element Array

```
sum = 0;
for (i=0; i<N; i++) sum += A[i];
printf(“Sum of array A is, %4d”,sum);
```

OR

```
sum = 0;
i = 0;
Rept: sum += A[i];
i++;
if (i < N) goto Rept
printf(“Sum of array A is, %4d”,sum);
```

```
sum = 0;
i = 0;
Rept: if (l >= N) goto Done
sum += A[i];
i++;
goto Rept
Done: printf(“Sum of array A is, %4d”,sum);
```
sum = 0;
i = 0;
Rept: sum += A[i];
i++;
if (i < N) goto Rept
printf(“Sum of array A is, %4d”, sum);

A:
.data
.word  5:20  # 20-element array of initial values of 5

.text
.globl main
main: li  $t0, 0  # $t0 <- sum
    li  $a0, 0  # $a0 <- array index
    li  $t9, 80  # array index upper bound in bytes
    la  $t8, A  # $t8 <- array beginning address
rept: add  $t1, $t8, $a0  # $t1 <- address of current array element
    lw  $t2, ($t1)  # $t2 <- value in the array
    add  $t0, $t0, $t2  # sum += A[i]
    addi  $a0, $a0, 4  # i++ in bytes
    blt  $a0, $t9, rept
move  $a0, $t0  # print the value of sum
    li  $v0, 1
    syscall
Common, readable approach: Use array as displacement in indirect addressing

```assembly
.data
A: .word 5:20  # 20-element array of initial values of 5

.text
.globl main
main:
  li $t0, 0  # $t0 <- sum
  li $a0, 0  # $a0 <- array index
  li $t9, 80 # array index upper bound in C context
rept:
  lw $t2, A($a0)  # $t2 <- value in the array
  add $t0, $t0, $t2 # sum += A[i]
  addi $a0, $a0, 4 # i++ in bytes
  blt $a0, $t9, rept

  move $a0, $t0  # print the value of sum
  li $v0, 1
  syscall
```