Operands and Addressing Modes

- Where is the data?
- Addresses as data
- Names and Values
- Indirection
Assembly Exercise

- Let’s write some assembly language programs

- Program #1: Write a function “isodd(int X)” which returns 1 if it’s argument “X” is odd and 0 otherwise

```
main:   addiu $a0,$0,37
        jal   isodd
        addiu $a0,$0,42
        jal   isodd
halt:   beq   $0,$0,halt
isodd:  andi  $v0,$a0,1
        jr    $31
```

The addiu instruction is used to load constants (i.e. isodd(37)), can this be done in other ways?

The function is implemented using only one instruction. How does “andi $Y,$X,1” determine that $X$ is odd?
Your Turn

• Program #2: A function “ones(int X)” that returns a count of the number of ones in its argument “X”
Last Time - “Machine” Language

32-bit (4-byte) ADD instruction:

```
0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0
```

op = R-type   Rs   Rt   Rd   func = add


But, most of us would prefer to write

```
add $3, $4, $2   (ASSEMBLER)
```

or, better yet,

```
a = b + c;   (C)
```
Revisiting Operands

• **Operands** – the variables needed to perform an instruction’s operation

• **Three types in the MIPS ISA:**
  
  - **Register:**
    
    ```
    add $2, $3, $4   # operands are the “Contents” of a register
    ```
  
  - **Immediate:**
    
    ```
    addi $2,$2,1   # 2\textsuperscript{nd} source operand is part of the instruction
    ```
  
  - **Register-Indirect:**
    
    ```
    lw $2, 12($28)   # source operand is in memory
    sw $2, 12($28)   # destination operand is memory
    ```

• **Simple enough, but is it enough?**
Common “Addressing Modes”

MIPS can do these with appropriate choices for Ra and const

- **Absolute (Direct):** `lw $8, 0x1000($0)`
  - Value = Mem[constant]
  - Use: accessing static data

- **Indirect:** `lw $8, 0($9)`
  - Value = Mem[Reg[x]]
  - Use: pointer accesses

- **Displacement:** `lw $8, 16($9)`
  - Value = Mem[Reg[x] + constant]
  - Use: access to local variables

- **Indexed:**
  - Value = Mem[Reg[x] + Reg[y]]
  - Use: array accesses (base+index)

- **Memory indirect:**
  - Value = Mem[Mem[Reg[x]]]
  - Use: access thru pointer in mem

- **Autoincrement:**
  - Value = Mem[Reg[x]]; Reg[x]++
  - Use: sequential pointer accesses

- **Autodecrement:**
  - Value = Reg[x]--; Mem[Reg[x]]
  - Use: stack operations

- **Scaled:**
  - Value = Mem[Reg[x] + c + d*Reg[y]]
  - Use: array accesses (base+index)

Argh! Is the complexity worth the cost? Need a cost/benefit analysis!
Memory Operands: Usage

Usage of different memory operand modes

© 2003 Elsevier Science (USA). All rights reserved.
Absolute (Direct) Addressing

• What we want:
  – The contents of a specific memory location

• Examples:

  “C”
  ```
  int x = 10;
  main() {
    x = x + 1;
  }
  ```

  “MIPS Assembly”
  ```
  main:   lw  $2,x
          addi $2,$2,1
          sw  $2,x
          jr  $31
  ```

  ```
  x: .word 10
  ```
  Allocates space for a single integer (4-bytes) and initializes its value to 10

• Caveats
  – In practice $gp is often used as a base address for variables
  – Can only address the first and last 32K of memory this way
  – Sometimes generates a two instruction sequence:

    ```
    lui  $1,xhighbits
    lw   $2,xlowbits($1)
    ```
An Aside: Let’s C

C is an ancestor to many languages commonly used today.
   {Algol, Fortran, Pascal} → C → C++ → Java

C was developed to write the operating system UNIX.

C is still widely used for “systems” programming

C’s developers were frustrated that the high-level languages
available at the time, lacked the expressiveness and
capabilities of assembly code necessary to write an OS.

The advantage of high-level languages is that they are
portable (i.e. not ISA specific).

C, thus, was an attempt to create a portable blend of a high-
level language and an assembler
C begat Java

C++ was envisioned to add Object-Oriented (OO) concepts on top of C
Java was envisioned to be more purely OO, and hide the details of Class/Method/Member implementation

For our purposes C is almost identical to JAVA except:

C has “functions”, whereas JAVA has “methods”.
C has explicit variables that contain the addresses of other variables or data structures in memory.
JAVA hides them under the covers.
C pointers

int i;       // simple integer variable
int a[10];   // array of integers (a is a pointer)
int *p;      // pointer to integer(s)

*(expression) is content of address computed by expression.

a[k] ≡ *(a+k)

a is a constant of type "int *

a[k] = a[k+1] ≡ *(a+k) = *(a+k+1)
Other Pointer Related Syntax

```c
int i;       // simple integer variable
int a[10];  // array of integers
int *p;     // pointer to integer(s)

p = &i;      // & means address of
p = a;       // no need for & on a
p = &a[5];   // address of 6th element of a
*p          // value of location pointed by p
*p = 1;      // change value of that location
*(p+1) = 1;  // change value of next location
p[1] = 1;    // exactly the same as above
p++;         // step pointer to the next element
```
Legal uses of Pointers

int i;       // simple integer variable
int a[10];  // array of integers
int *p;     // pointer to integer(s)

So what happens when
p = &i;
What is value of p[0]?
What is value of p[1]?
C Pointers vs. object size

int i;          // simple integer variable
int a[10];     // array of integers
int *p;        // pointer to integer(s)

Does "p++" really add 1 to the pointer?
NO! It adds 4. Why 4?

char *q;
...
q++;  // really does add 1
void clear1(int array[], int size) {
    for (int i=0; i<size; i++)
        array[i] = 0;
}

void clear2(int array[], int size) {
    for (int *p = &array[0]; p < &array[size]; p++)
        *p = 0;
}

void clear3(int *array, int size) {
    int *end = array + size;
    while (array < end)
        *array++ = 0;
}
Pointer summary

• In the “C” world and in the “machine” world:
  – a pointer is just the address of an object in memory
  – size of pointer is fixed regardless of size of object
  – to get to the next object increment by the object’s size in bytes
  – to get the the i\textsuperscript{th} object add i*\texttt{sizeof(object)}

• More details:
  – int R[5] ≡ R is int* constant address of 20 bytes storage
  – R[i] ≡ *(R+i)
  – int *p = &R[3] ≡ p = (R+3) (p points 12 bytes after R)
Indirect Addressing

• What we want:
  - The contents of a memory location held in a register

• Examples:

  "C"
  ```
  int x = 10;
  main() {
    int *y = &x;
    *y = 2;
  }
  ```

  "MIPS Assembly"
  ```
  main:  ori  $2,$0,x
          addi $3,$0,2
          sw   $3,0($2)
          jr   $31
  ```

  x:   .word   10

• Caveats
  - You must make sure that the register contains a valid address
    (double, word, or short aligned as required)
Displacement Addressing

• What we want:
  - The contents of a memory location relative to a register

• Examples:

  “C”
  ```c
  int a[5];
  main() {
    int i = 3;
    a[i] = 2;
  }
  ```

  “MIPS Assembly”
  ```mips
  main:   addi $2,$0,3
          addi $3,$0,2
          sll  $1,$2,2
          sw   $3,a($1)
          jr   $31
  
  a:      .space   5
  ```

• Caveats
  - Must multiply (shift) the “index” to be properly aligned
Displacement Addressing: Once More

• What we want:
  – The contents of a memory location relative to a register

• Examples:

  “C”

  ```c
  struct p {
    int x, y;
  }

  main() {
    p.x = 3;
    p.y = 2;
  }
  ```

  “MIPS Assembly”

  ```assembly
  main:  ori $1,$0,p
          addi $2,$0,3
          sw  $2,0($1)
          addi $2,$0,2
          sw  $2,4($1)
          jr  $31
  ```

  Allocates space for 2 uninitialized integers (8-bytes)

• Caveats
  – Constants offset to the various fields of the structure
  – Structures larger than 32K use a different approach
There are little tricks that come into play when compiling conditional code blocks. For instance, the statement:

```c
if (y > 32) {
    x = x + 1;
}
```

compiles to:

```assembly
lw   $24, y
ori  $15, $0, 32
slt  $1, $15, $24
beq  $1, $0, Lendif
lw   $24, x
addi $24, $24, 1
sw   $24, x
Lendif:
```
C/Assembly Translation: Loops

C code:

while (expr) {
    STUFF
}

MIPS assembly:

Lwhile:
    (compute expr in $rx)
    beq $rX,$0,Lendw
    (compile STUFF)
    beq $0,$0,Lwhile

Lendw:

Alternate MIPS assembly:

beq $0,$0,Ltest
Lwhile:
    (compile STUFF)
Ltest:
    (compute expr in $rx)
    bne $rX,$0,Lwhile
Lendw:

Compilers spend a lot of time optimizing in and around loops.
- moving all possible computations outside of loops
- unrolling loops to reduce branching overhead
- simplifying expressions that depend on “loop variables”
C/Assembly Translation: For Loops

• Most high-level languages provide loop constructs that establish and update an iteration variable, which is used to control the loop’s behavior.

C code:

```c
int sum = 0;
int data[10] = {1,2,3,4,5,6,7,8,9,10};

int i;
for (i=0; i<10; i++) {
    sum += data[i]
}
```

MIPS assembly:

```assembly
sum:
    .word 0x0
data:
    .word 0x1, 0x2, 0x3, 0x4, 0x5
    .word 0x6, 0x7, 0x8, 0x9, 0xa
    add $30,$0,$0
Lfor:
    lw $24,sum($0)
    sll $15,$30,2
    lw $15,data($15)
    addu $24,$24,$15
    sw $24,sum
    add $30,$30,1
    slt $24,$30,10
    bne $24,$0,Lfor
Lendfor:
```

Next Time

- Pseudo instructions
- More C idioms
- Calling procedures
- Recursion