Operands and Addressing Modes

- Where is the data?
- Addresses as data
- Names and Values
- Indirection
Just enough C

For our purposes C is almost identical to JAVA except:

C has “functions”, JAVA has “methods”.
    function == method without “class”.
    A global method.
C has “pointers” explicitly. JAVA has them but hides them under the covers.
**C pointers**

```c
int i;       // simple integer variable
int a[10];  // array of integers
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] EQUIV *(a+k) = *(a+k+1)
```
Legal uses of C Pointers

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

```c
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

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

    char *q;
    ...
    q++;  // really does add 1
Clear123

```c
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 *arrayend = array + size;
    while(array < arrayend) *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:
  – \texttt{int R[5] \rightarrow R} is \texttt{int*} constant address of 20 bytes
  – \texttt{R[i] \rightarrow *(R+i)}
  – \texttt{int *p = \&R[3] \rightarrow p = (R+3)} (p points 12 bytes after R)
Last Time - “Machine” Language

32-bit (4-byte) ADD instruction:

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\text{op} = \text{R-type} \quad \text{Rs} \quad \text{Rt} \quad \text{Rd} \quad \text{func} = \text{add}
\]

Means, to MIPS, \( \text{Reg}[3] = \text{Reg}[4] + \text{Reg}[2] \)

But, most of us would prefer to write

\[
\text{add} \; \$3, \; \$4, \; \$2 \quad \text{(ASSEMBLER)}
\]

or, better yet,

\[
a = b+c; \quad \text{(C)}
\]
Revisiting Operands

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

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

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

- **Absolute:** `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 Addressing

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

• Examples:

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

  “MIPS Assembly”
  ```
  .data
  .global x
  x: .word 10
  .text
  .global main
  main:
    lw $2,x($0)
    addi $2,$2,1
    sw $2,x($0)
  ```

• Caveats
  – In practice $gp is used instead of $0
  – 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)
    ```
Indirect Addressing

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

• Examples:

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

  “MIPS Assembly”
  ```mips
  .data
  .global x
  x: .word 10

  .text
  .global main
  main:
    la $2,x
    addi $3,$0,2
    sw $3,0($2)
  ```

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

“la” is not a real instruction, it’s a convenience pseudoinstruction that constructs a constant via either a 1 instruction or 2 instruction sequence
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
  .data
  .global a
  a: .space 20
  .text
  .global main
  main:
    addi $2,$0,3
    addi $3,$0,2
    sll $1,$2,2
    sw $3,a($1)
  ```

• 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”
  struct p {
    int x, y;
  }

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

  “MIPS Assembly”

  .data
  .global p
  p: .space 8

  .text
  .global main
  main:
    la $1, p
    addi $2, $0, 3
    sw $2, 0($1)
    addi $2, $0, 2
    sw $2, 4($1)

• Caveats
  – Constants offset to the various fields of the structure
  – Structures larger than 32K use a different approach
Conditionals

C code:
if (expr) {
    STUFF
}
else {
    STUFF2
}

C code:
if (expr) {
    STUFF1
} else {
    STUFF2
}

MIPS assembly:
(compute expr in $rx)
beq $rx, $0, Lendif

(compile STUFF)
Lendif:

MIPS assembly:
(compute expr in $rx)
beq $rx, $0, Lelse

(compile STUFF1)
beq $0, $0, Lendif
Lelse:

(compile STUFF2)
Lendif:

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:
```
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:
```
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”
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:

```mips
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

• We’ll write some real assembly code
• Play with a simulator