Computer Organization and Design

Addressing Modes

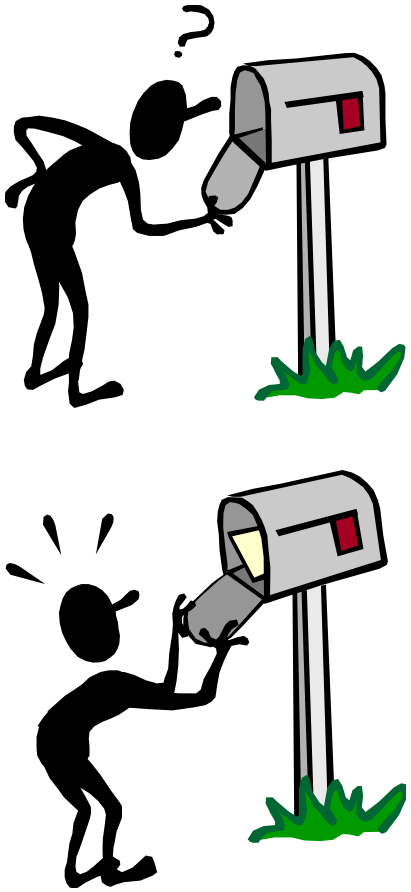
Henry Fuchs

Slides adapted from Montek Singh, who adapted them
from Leonard McMillan and from Gary Bishop
Back to McMillan & Chris Terman, MIT 6.004 1999

Tuesday, Feb. 10, 2015

Lecture 6

Operands and Addressing Modes



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

Reading: Ch. 2.3, 2.14

Pointers and Arrays in C

Operands in memory and their addresses

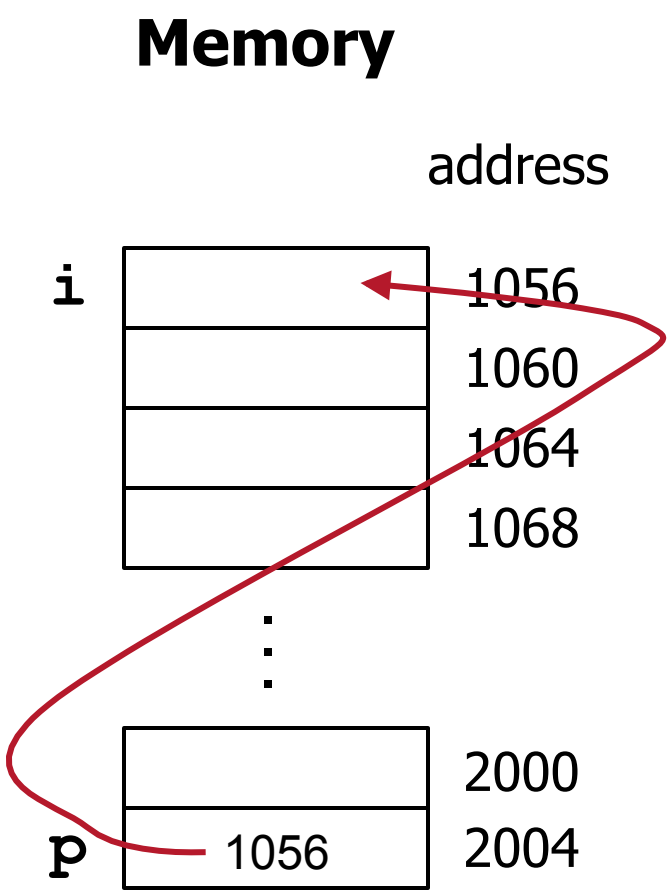
C vs. Java

- For our purposes C is almost identical to Java except:
  - C has “functions”, JAVA has “methods”
    - function == method without “class”
    - i.e., a global method
  - C has “pointers” explicitly
    - Java has them (called “references”) but hides them under the covers
    - JVM takes care of handling pointers, so the programmer doesn’t have to
- C++ is sort of in-between C and Java

➔In this class, we will see how pointers/references are implemented at the assembly language level

What is a "pointer" in C?

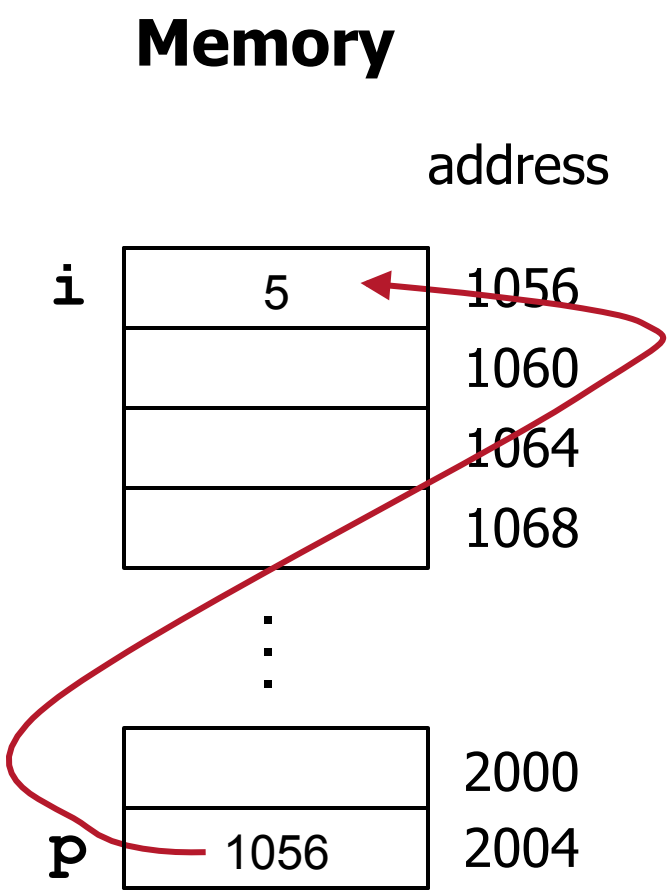
- A pointer is an explicit memory address
- Example
  - `int i`
    - `i` is an integer variable
    - located at, say, address 1056
  - `int *p`
    - `p` is a variable that "points to" an integer
    - `p` is located at, say, address 2004
  - `p = &i`
    - the **value in `p`** is now equal to the **address of variable `i`**
    - i.e., the value stored in Mem[2004] is 1056 (the location of `i`)



Referencing and Dereferencing

- Referencing an object means
  - … taking its address and assigning it to a pointer variable (e.g., `&i`)
- Dereferencing a pointer means
  - … going to the memory address pointed to by the pointer, and accessing the value there (e.g., `*p`)
- Example

```
int i;   // i is an int variable
int *p;  // p is a pointer to int
p = &i;  // referencing i
         // p is assigned 1056
*p = 5;  // dereference p
         // i assigned 5
```

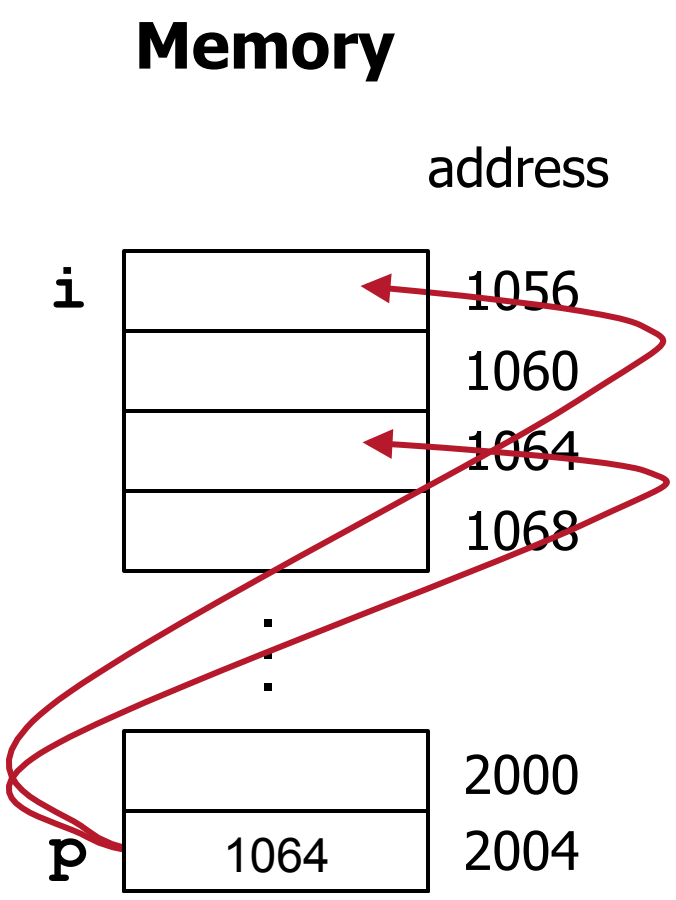


Pointer expressions and arrays

- Dereferencing could be done to an expression
  - So, not just `*p`, but can also write `*(p+400)`
    - accesses memory location that is 400th int after `i`
- Arrays in C are really pointers underneath!
  - `int a[10];  // array of integers`
  - `a` itself simply refers to the address of the *start* of the array
  - `a` is the same as `&a[0]`
  - `a` is a constant of type "`int *`"
  - `a[0]` is the same as `*a`
  - `a[1]` is the same as `*(a+1)`
  - `a[k]` is the same as `*(a+k)`
  - `a[j] = a[k];` is the same as `*(a+j) = *(a+k);`

Pointer arithmetic and object size

- IMPORTANT: Pointer expressions automatically account for the size of object pointed to
  - Example 1
    - if p is of type "int *"
    - and an int is 4 bytes long
    - if `p` points to address 1056, `(p=p+2)` will point to address 1064
    - C compiler automatically does the multiply-by-4
    - BUT… in assembly, the programmer will have to explicitly do the multiply-by-4
  - Example 2
    - `char *q;`
    - `q++; // really does add 1`



Pointer examples


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

p = &i;      // & means address of
p = a;       // a means &a[0]
p = &a[5];   // address of 6th element of a
*p           // value at location pointed by p
*p = 3;      // change value at that location
*(p+1) = 7; // change value at next location
p[1] = 7;    // exactly the same as above
p++;         // step pointer to the next element
```

Pointer pitfalls


```
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]`?

➔ Very easy to exceed bounds (C has no bounds checking)

Iterating through an array

- 2 ways to iterate through an array
  - using array indices

```
void clear1(int array[], int size) {
  for(int i=0; i<size; i++)
    array[i] = 0;
}
```

  - using pointers

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

  - or, also using pointers, but more concise (more cryptic!)

```
void clear3(int *array, int size) {
  int* arrayend;  arrayend = array + size;
  while(array < arrayend) {*array = 0; array=array+1 }
}
```

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 itself is fixed regardless of size of object
  - to get to the next object:
    - in machine code: increment pointer by the object's size in bytes
    - in C: increment pointer by 1
  - to get the ith object:
    - in machine code: add `i*sizeof(object)` to pointer
    - in C: add `i` to pointer
- Examples:
  - int R[5]; // 20 bytes storage
  - R[i] is same as *(R+i)
  - int *p = &R[3] is same as p = (R+3)
    (p points 12 bytes after start of R)

Addressing Modes

What are all the different ways to specify an operand?

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     # 2nd 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, such as looping thru an array

Autodecrement:
- Value = Reg[X]--; Mem[Reg[x]]
- Use: stack operations
- (Decrements first to match autoincrement)

Scaled:
- Value = Mem[Reg[x] + c + d*Reg[y]]
- Use: array accesses (base+index) and 2D arrays

Pentium & some others have more of these addressing modes
Is the complexity worth the cost? Need a cost/benefit analysis.

Relative Popularity of Addressing Modes

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

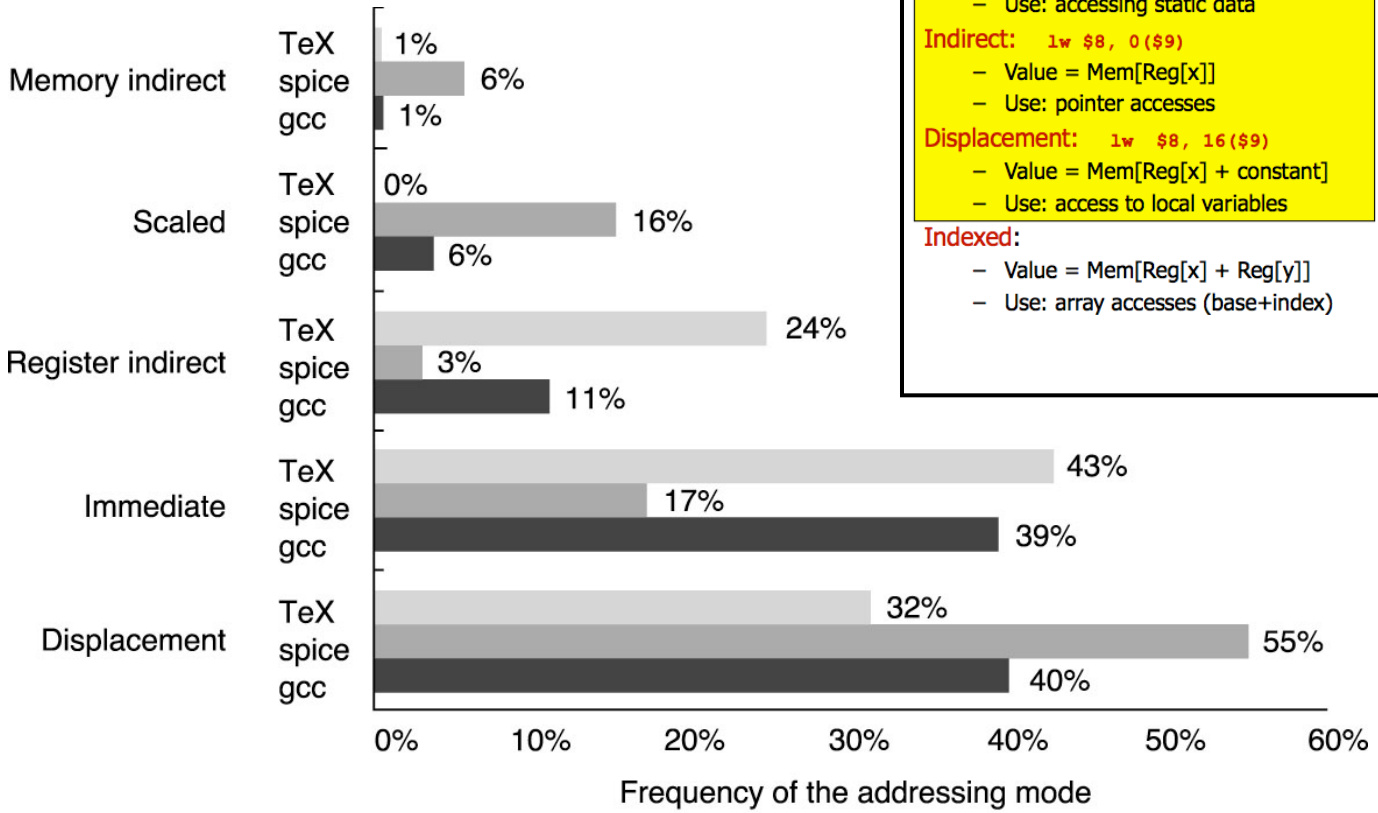
Indexed:
- Value = Mem[Reg[x] + Reg[y]]
- Use: array accesses (base+index)

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

Autoincrement:
- Value = Mem[Reg[x]]; Reg[x]++
- Use: sequential pointer accesses, such as looping thru an array

Autodecrement:
- Value = Reg[X]--; Mem[Reg[x]]
- Use: stack operations
- (Decrements first to match autoincrement)

Scaled:
- Value = Mem[Reg[x] + c + d*Reg[y]]
- Use: array accesses (base+index) and 2D arrays



From Hennessy & Patterson

Absolute (Direct) Addressing

- What we want:
  - Contents of a specific memory location, i.e. at a given address
- Example:

"C"

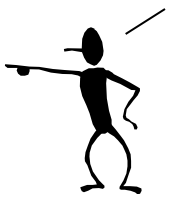
```
int x = 10;

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)
```



Allocates space for a single integer (4-bytes) and initializes its value to 10

- 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)
```

Absolute (Direct) Addressing: More detail

"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)
```

"After Compilation"

```
.data 0x0100
.global x
x: .word 10

.text
.global main
main:
        lw   $2,0x100($0)
        addi $2,$2,1
        sw   $2,0x100($0)
```

- Assembler replaces "**x**" by its address
  - e.g., here the data part of the code (**.data**) starts at 0x100
  - **x** is the first variable, so starts at 0x100

Indirect Addressing

- What we want:
  - The contents at a memory address held in a register

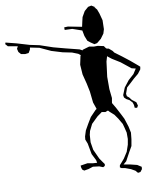
"la" is not a real instruction, It's a convenient *pseudoinstruction* that constructs a constant via either a 1 instruction or 2 instruction sequence

- Examples:

"C"

```
int x = 10;

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



Store 2 in adr that y is pointing to

"MIPS Assembly"

```
.data
.global x
x: .word   10

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

```
ori   $2,$0,x

lui   $2,xhighbits
ori   $2,$2,xlowbits
```

Note that "load adr" does not reference memory

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

Note on `la` pseudoinstruction

- **`la` is a pseudoinstruction: `la $r, x`**
  - stands for "load the address of" variable *x* into register *r*
  - not an actual MIPS instruction
  - but broken down by the assembler into actual instructions
    - if address of x is small (fits within 16 bits), then a single **`ori`**
    - if address of x is larger, use the **`lui + ori`** combo

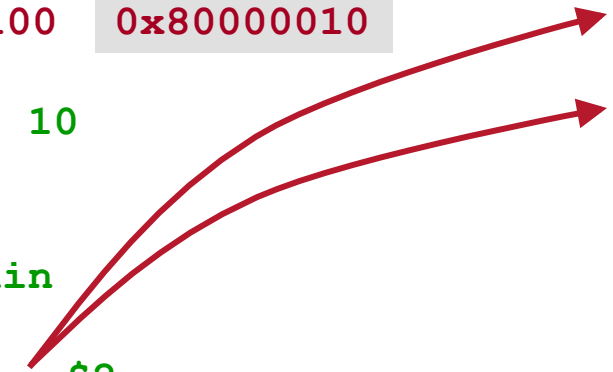
"MIPS Assembly"

```
.data 0x0100   0x80000010
.global x
x: .word   10

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

```
ori   $2,$0,0x100

lui   $2,0x8000
ori   $2,$2,0x0010
```



Displacement Addressing

- What we want:
  - contents of a memory location at an offset relative to a register
    - the address that is the sum of a constant and a register value
- Examples:

"C"

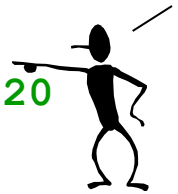
```
int a[5];

main() {
    int i = 3;
    a[i] = 2;
}
```

"MIPS Assembly"

```
.data 0x0100
.global a
a: .space   20

.text
.global main
main:
    addi $2,$0,3   // i in $2
    addi $3,$0,2
    sll  $1,$2,2   // i*4 in $1
    sw   $3,a($1)
```



Allocates space for a 5 uninitialized integers (20-bytes)

- Caveats
  - Must multiply (shift) the "index" to be properly aligned

Displacement Addressing: 2nd example

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

"C" structures

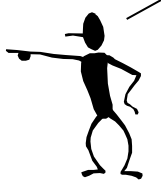
```
struct p {
   int x, y; }

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

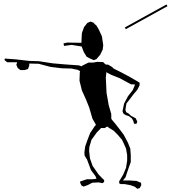
"MIPS Assembly"

```
.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)
```



Allocates space for 2 uninitialized integers (8-bytes)



Offset by 0 of value of register storing p's adr.

Offset by 4 bytes of value of register storing p's adr.

- Note
  - offsets to the various fields within a structure are constants known to the assembler/compiler

Assembly Coding Templates

For common C program fragments

Conditionals: `if-else`

C code:

```
if (expr) {
    STUFF
}
```

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:

```
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: `while`

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”

Loops: for

- 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:

```
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:

```
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 Class

- We'll write some real assembly code
- Play with a simulator
- Bring your laptop computers!