**MIPS Assembly Language Examples**

**Preliminaries**

MIPS has 32 "general purpose registers". As far as the hardware is concerned, they are all the same, with the sole exception of register 0, which is hardwired to the value 0.

As we'll see later, there are software **conventions** that restrict the use of registers - an application will run correctly if it follows these conventions, but may fail if it doesn't and it interacts with any other software. (Since all applications interact with the operating system, unless you find a way to load your application onto bare hardware, you need to follow the conventions.)  For the moment, we'll only roughly follow some of the conventions. For each example below, we'll assume that the PC points to the first instruction in our code, that we can use the registers called (in assembly language) $t0-$t9 as we please, and that register $gp points to an area in memory that we can use to hold program variables.

These examples take advantage of the full MIPS instruction set. The tool we will be using, [Cebollita](http://www.cs.washington.edu/homes/dugan/cebollita/docs/), models a processors that implements only a subset. (This means that if you try to assemble these programs in Cebollita, some will not work.)

**Data / Memory Layout**

We'll assume all the examples, which are fragments of C programs, include the following:

// none of these allocate any storage

#define MAX\_SIZE 256

#define IF(a) if (a) {

#define ENDIF }

typedef struct {

unsigned char red; // 'unsigned char' is an unsigned, 8-bit int

unsigned char green;

unsigned char blue;

unsigned char alpha;

} RGBa;

// these allocate storage

int i;

int N = 20;

char prompt[] = "Enter an integer:";

int A[MAX\_SIZE];

int\* pBArray;

int BSize;

RGBa background = {0xff, 0xff, 0xff, 0x0};

and further that code has already been executed that initializes pBArray to point to some integer array and to set BSize to the size of that array. (Depending on the C compiler), **the memory layout** [looks like this](http://www.cs.washington.edu/education/courses/cse378/03wi/lectures/mem-layout.jpg).

**A Simple Expression**

C code:

i = N\*N + 3\*N

"Unoptimized":   
*(Note: There are some small disagreements in the syntax of assembler between SPIM, which is used in the book, and Cebollita, which is the tool we will be using. I have tried to follow the conventions of Cebollita here.)*

lw $t0, 4($gp) # fetch N

mult $t0, $t0, $t0 # N\*N

lw $t1, 4($gp) # fetch N

ori $t2, $zero, 3 # 3

mult $t1, $t1, $t2 # 3\*N

add $t2, $t0, $t1 # N\*N + 3\*N

sw $t2, 0($gp) # i = ...

"Optimized":

lw $t0, 4($gp) # fetch N

add $t1, $t0, $zero # copy N to $t1

addi $t1, $t1, 3 # N+3

mult $t1, $t1, $t0 # N\*(N+3)

sw $t1, 0($gp) # i = ...

**Array Expression and Inter-Statement Optimizations**

C code:

A[i] = A[i/2] + 1;

A[i+1] = -1;

"Unoptimized":

# A[i] = A[i/2] + 1;

lw $t0, 0($gp) # fetch i

srl $t0, $t0, 1 # i/2

addi $t1, $gp, 28 # &A[0]

sll $t0, $t0, 2 # turn i/2 into a byte offset (\*4)

add $t1, $t1, $t0 # &A[i/2]

lw $t1, 0($t1) # fetch A[i/2]

addi $t1, $t1, 1 # A[i/2] + 1

lw $t0, 0($gp) # fetch i

sll $t0, $t0, 2 # turn i into a byte offset

addi $t2, $gp, 28 # &A[0]

add $t2, $t2, $t0 # &A[i]

sw $t1, 0($t2) # A[i] = ...

# A[i+1] = -1;

lw $t0, 0($gp) # fetch i

addi $t0, $t0, 1 # i+1

sll $t0, $t0, 2 # turn i+1 into a byte offset

addi $t1, $gp, 28 # &A[0]

add $t1, $t1, $t0 # &A[i+1]

addi $t2, $zero, -1 # -1

sw $t2, 0($t1) # A[i+1] = -1

"Optimized":

# A[i] = A[i/2] + 1;

lw $t0, 0($gp) # fetch i

srl $t1, $t0, 1 # i/2

sll $t1, $t1, 2 # turn i/2 into a byte offset (\*4)

add $t1, $gp, $t1 # &A[i/2] - 28

lw $t1, 28($t1) # fetch A[i/2]

addi $t1, $t1, 1 # A[i/2] + 1

sll $t2, $t0, 2 # turn i into a byte offset

add $t2, $t2, $gp # &A[i] - 28

sw $t1, 28($t2) # A[i] = ...

# A[i+1] = -1;

addi $t1, $zero, -1 # -1

sw $t1, 32($t2) # A[i+1] = -1

**#define and if Statement**

C code pre-cfront:

IF (i < N)

A[i] = 0;

ENDIF

C code post-cfront:

if (i<N) {

A[i] = 0;

}

MIPS assembler:

lw $t0, 0($gp) # fetch i

lw $t1, 4($gp) # fetch N

slt $t1, $t0, $t1 # set $t1 to 1 if $t0 < $t1, to 0 otherwise

beq $t1, $zero, skip # branch if result of slt is 0 (i.e., !(i<N))

sll $t0, $t0, 2 # i as a byte offset

add $t0, $t0, $gp # &A[i] - 28

sw $zero, 28($t0) # A[i] = 0

skip:

**And Mask**

C code:

background.blue = background.blue \* 2; // Note: overflow...

MIPS Assembler:

lw $t0, 1060($gp) # fetch background

andi $t1, $t0, 0xff00 # isolate blue

sll $t1, $t1, 2 # times 2

andi $t1, $t1, 0xff00 # get rid of overflow

lui $t2, 0xffff # $t2 = 0xffff0000

ori $t2, $t2, 0x00ff # $t2 = 0xffff00ff

and $t0, $t0, $t2 # get rid of old value of blue

or $t0, $t0, $t1 # new value

sw $t0, 1060($gp) # background = ...

**Or Mask**

C code:

// set N to the smallest odd no less than N

if ( N%2 == 0 ) N++;

MIPS Assembler:

lw $t0, 4($gp) # fetch N

ori $t0, $t0, 1 # turn on low order bit

sw $t0, 4($gp) # store result in N

**switch Statement**

C code:

switch (i) {

case 0: A[0] = 0;

break;

case 1:

case 2: A[1] = 1;

break;

default: A[0] = -1;

break;

}

For this example, assume the compiler has generated a *branch table* and stored it after background in memory (i.e., starting at offset 1064 from $gp). The branch table is initialized to hold in successive locations the absolute addresses of the instructions at labels is0, is1, and is2.

MIPS Assembler:

lw $t0, 0($gp) # fetch i

bltz $t0, def # i<0 -> default

slti $t1, $t0, 3 # i<3?

beq $t1, $zero, def # no, -> default

sll $t0, $t0, 2 # turn i into a byte offset

add $t2, $t0, $gp

lw $t2, 1064($t2) # fetch the branch table entry

jr $t2 # go...

is0: sw $zero, 28($gp) # A[0] = 0

j done

is1:

is2: addi $t0, $zero, 1 # = 1

sw $t0, 32($gp) # A[1] = 1

j done

def: addi $t0, $zero, -1 # = -1

sw $t0, 28($gp) # A[0] = -1

j done

done:

**for Loop**

C code:

for (i=0; i<N; i++) {

A[i] = MAX\_SIZE;

}

MIPS Assembler

add $t0, $gp, $zero # &A[0] - 28

lw $t1, 4($gp) # fetch N

sll $t1, $t1, 2 # N as byte offset

add $t1, $t1, $gp # &A[N] - 28

ori $t2, $zero, 256 # MAX\_SIZE

top:

sltu $t3, $t0, $t1 # have we reached the final address?

beq $t3, $zero, done # yes, we're done

sw $t2, 28($t0) # A[i] = 0

addi $t0, $t0, 4 # update $t0 to point to next element

j top # go to top of loop

done:

# NOTE: We have not updated i in memory!