1 Common MIPS Uses

Comment each snippet with what the snippet does. Assume that there is an array, int pi[6] = {3, 1, 4, 1, 5, 9}, which is stored beginning at memory address 0xBFFFFF00, and a linked list struct (as defined below), struct ll raspberry;, which is stored beginning at memory adddress 0xABCD0000. $s0 then contains pi’s address, 0xBFFFFF00, and $s1 contains raspberry’s addresss, 0xABCD0000.

struct ll {
    int val;
    struct ll* next;
}

pi[1] = pi[0] + pi[2];

# Array Reading/Writing
lw $t0 0($s0) # t0 = pi[0] = 3;
lw $t1 8($s0) # t1 = pi[2] = 4;
addu $t2 $t0 $t1 # t2 = t0 + t1 = 7;
sw $t2 4($s0) # pi[1] = t2 = 7;

raspberry->val += 1;
raspberry->next->val += 1;

# Struct Accessing
lw $t0 0($s1) # t0 = raspberry->val;
addiu $t0 $t0 1 # t0 += 1;
sw $t0 0($s1) # raspberry->val = t0;

# raspberry->next->val += 1;
lw $s2 4($s1) # s2 = raspberry->next;
lw $t1 0($s2) # t1 = raspberry->next->val;
addiu $t1 $t1 1 # t1 += 1;
sw $t1 0($s2) # raspberry->next->val = t1;

if (a0 != 0)
— a0 += -2;
else
— a0 += 4
a0 += 4;

# If Statements
beq $a0 $0 Else # if (a0 != 0)
If:
    addiu $a0 $a0 -2 # a0 += -2;
j End
Else:
    addiu $a0 $a0 3 # else {a0 += 3;
    addiu $a0 $a0 1 # a0 += 1;}
End:
    addiu $a0 $a0 4 # a0 += 4;

int i;
int sum = 0;
for (i = 0; i < 6; i++)
— sum += pi[i];

# For Loop
addu $t0 $0 $0 # t0 = 0;
addiu $t1$0 6 # t1 = 6;
addu $t2 $0 $0 # t2 = 0;

L1: beq $t0 $t1 L2 # while (t0 != t1)
sll $t3 $t0 2 # t3 = t0 * 4;
addu $s2 $t3 $s0 # s2 = t3 + s0
# s2 = 0xBFFFFF00 offset t3 bytes
lw $t4 0($s2) # t4 = pi[t0];
addu $t2 $t2 $t4 # sum += t4;
addiu $t0 $t0 1 # t0 += 1;
L2: # end of loop

2 Translating between C and MIPS

Translate between the C and MIPS code. You may want to use the MIPS Green Sheet as a reference. We show you how the different variables map to registers – you don’t have to worry about the stack or any memory-related issues.
### 3 MIPS Addressing

- We have several addressing modes to access memory (immediate not listed):

  (a) **Base displacement addressing:** Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)

  (b) **PC-relative addressing:** Uses the PC (actually the current PC plus four) and adds the I-value of the instruction (multiplied by 4) to create an address (used by I-format branching instructions like beq, bne)

  (c) **Pseudodirect addressing:** Uses the upper four bits of the PC and concatenates a 26-bit value from the instruction (with implicit 00 lowest bits) to make a 32-bit address (used by J-format instructions)

  (d) **Register Addressing:** Uses the value in a register as a memory address (jr)

1. You need to jump to an instruction that $2^{28} + 4$ bytes higher than the current PC. How do you do it? Assume you know the exact destination address at compile time. (Hint: you need multiple instructions)

   The jump instruction can only reach addresses that share the same upper 4 bits as the PC. A jump $2^{28} + 4$ bytes away would require changing the fourth highest bit, so a jump instruction is not sufficient. We must manually load our 32 bit address into a register and use jr.

   ```assembly
   lui $at {upper 16 bits of Foo}
   ori $at $at {lower 16 bits of Foo}
   jr $at
   ```

2. You now need to branch to an instruction $2^{17} + 4$ bytes higher than the current PC, when $t0$ equals 0. Assume that we’re not jumping to a new $2^{28}$ byte block. Write MIPS to do this.

   The largest address a branch instruction can reach is PC + 4 + SignExtImm. The immediate field is 16 bits and signed, so the largest value is $2^{15} - 1$ words, or $2^{17} - 4$ Bytes. Thus, we cannot use a branch instruction to reach our goal, but by the problem’s assumption, we can use a jump. Assuming we’re jumping to label Foo

   ```assembly
   ... beq $s0, $0, Ret0
   addiu $t2, $0, 1
   beq $s0, $t2, Ret1
   addiu $s0, $s0, -2
   Loop: beq $s0, $0, RetF
   addu $s1, $t0, $t1
   addiu $t0, $t1, 0
   addiu $t1, $s1, 0
   addiu $s0, $s0, -1
   j Loop
   Ret0: addiu $v0, $0, 0
   j Done
   Ret1: addiu $v0, $0, 1
   j Done
   RetF: addu $v0, $0, $s1
   Done: ...
   ```
bne $t0 $0 DontJump
j Foo
DontJump: ...

3. Given the following MIPS code (and instruction addresses), fill in the blank fields for the following instructions (you’ll need your green sheet):

```
0x002cff00: loop: addu $t0, $t0, $t0 | 0 | 8 | 8 | 8 | 0 | 0x21 |
0x002cff04: jal foo | 3 | 0xc0001 |
0x002cff08: bne $t0, $zero, loop | 5 | 8 | 0 | -3 = 0xfffd |
... 0x00300004: foo: jr $ra $ra=__0x002cff08___
```

4 MIPS Calling Conventions

1. How should $sp be used? When do we add or subtract from $sp?

   $sp points to a location on the stack to load or store into. Subtract from $sp before storing, and add to $sp after restoring.

2. Which registers need to be saved or restored before using jr to return from a function?

   All $s* registers that were modified during the function must be restored to their value at the start of the function.

3. Which registers need to be saved before using jal?

   $ra, and all $t*, $a*, and $v* registers if their values are needed later after the function call.

4. How do we pass arguments into functions?

   $a0, $a1, $a2, $a3 are the four argument registers.

5. What do we do if there are more than four arguments to a function?

   Use the stack to store additional arguments.

6. How are values returned by functions?

   $v0 and $v1 are the return value registers.
5 Writing MIPS Functions

Here is a general template for writing functions in MIPS:

```mips
FunctionFoo:   # PROLOGUE
   # begin by reserving space on the stack
   addiu $sp, $sp, -FrameSize

   # now, store needed registers
   sw $ra, 0($sp)
   sw $s0, 4($sp)
   ...
   # BODY
   ...
   # EPILOGUE
   # restore registers
   lw $s0 4($sp)
   lw $ra 0($sp)

   # release stack spaces
   addiu $sp, $sp, FrameSize

   # return to normal execution
   jr $ra
```

Translate the following C code for a recursive function into a callable MIPS function.

```c
// Finds the sum of numbers 0 to N
int sum_numbers(int N) {
    int sum = 0

    if (N==0) {
        return 0;
    } else {
        return N + sum_numbers(N - 1);
    }
}
```

```mips
RecursiveSum:
   addiu $sp, $sp, -8
   sw $ra, 4($sp)
   sw $a0, 0($sp)
   li $v0, 0
   beq $a0, $0, Ret
   addiu $a0, $a0, -1
   jal RecursiveSum
   lw $a0, 0($sp)
   addu $v0, $v0, $a0
   Ret:
   lw $ra, 4($sp)
   addiu $sp, $sp, 8
   jr $ra
```