1 Common MIPS Uses

Comment each snippet with what the snippet does. Assume that there is an array, int cash[6] = {1, 2, 3, 4, 5, 6}, which is stored beginning at memory address 0xBFFFFF00, and a linked list struct (as defined below), struct ll_node* money;, which is stored beginning at memory address 0xABCD0000. $s0 then contains cash’s address, 0xBFFFFF00, and $s1 contains money’s address, 0xABCD0000. Assume all snippets are INDEPENDENT.

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

# cash[1] = cash[0] + cash[2];
# Array Reading/Writing
lw $t0 0($s0) # t0 = cash[0] = 1;
lw $t1 8($s0) # t1 = cash[2] = 3;
addu $t2 $t0 $t1 # t2 = t0 + t1 = 4;
sw $t2 4($s0) # cash[1] = t2 = 4;
# What does cash look like at the end
# cash = {1, 4, 3, 4, 5, 6}

# money->val += 1;
# money->next->val += 1;
# Struct Accessing
lw $t0 0($s1) # t0 = money->val;
addiu $t0 $t0 1 # t0 += 1;
sw $t0 0($s1) # money->val = t0;
# money->next->val += 1;
lw $s2 4($s1) # s2 = money->next;
lw $t1 0($s2) # t1 = money->next->val;
addiu $t1 $t1 1 # t1 += 1;
sw $t1 0($s2) # money->next->val = t1;

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

2 Translating between C and MIPS

Translate between the C and MIPS code. You may want to use the MIPS Green (money) 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.

int i, sum = 0;
for (i = 0; i < 6; i++)
    sum += cash[i];
# For Loop
addu $t0 $0 $0 # t0 = 0; i
addiu $t1 $0 6 # t1 = 6; length
addu $t2 $0 $0 # t2 = 0; sum
L1: beq $t0 $t1 L2 # while (t0 != t1)
sll $t3 $t0 2 # t3 = t0 * 4; index
addu $s2 $t3 $s0 # s2 = t3 + s0
# s2 = 0xBFFFFF00 offset t3 bytes
lw $t4 0($s2) # t4 = cash[t0];
addu $t2 $t2 $t4 # sum += t4;
addu $t0 $t0 1 # t0 += 1;
L2: # end of loop
3 MIPS Addressing

- We have several **addressing modes** to access memory (understand what are the range of addresses that we can move to for each mode from PC):

  1. **Base displacement addressing**: Adds an immediate to a register value to create a memory address (used for lw, lh, lw, sw, sh, sb). e.g. lw $s0, 0($a1), sb $s0, 3($a1)

  2. **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 instructions like beq, bne. The labels get replaced by an immediate that indicates # instructions to move)

  3. **Pseudodirect addressing**: Uses the upper four bits of the PC and concatenates a 26-bit value from the instruction (with implicit 00 lowest bits, can you see why we have implicit 00 bits?) to make a 32-bit address (used by J-format instructions)

  4. **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.

   ```
elui $at {upper 16 bits of Foo}
or $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
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

When writing a function in MIPS, there are several things we must do to make sure we have good practice and code writing! First, there are several registers that must be the same before and after a function call:

- $sp
- $ra
- $s0 - $s7

When we say they must stay the same, we can still use them, but we should save their values before we change them and put them back when we’re done! This is because functions that call other functions (caller) can assume that these registers will be “untouched” by the function that is called (callee).

Now, in order to save these registers, we should save them to the memory, but you might be asking, where???

This is why we have the stack pointer! We will store things directly into the memory, and load them back when we’re done.

In order to make more room to store data, we move the stack pointer DOWN and save upwards, then load upwards and move the stack pointer back UP, e.g.:

```
addiu $sp, $sp, -8 # Move the stack pointer down  
sw $s0, 0($sp) # Save the required registers  
sw $ra, 4($sp)  
...  
lw $s0, 0($sp) # Load back the required registers  
lw $ra, 4($sp)  
addiu $sp, $sp, 8 # Move the stack pointer down  
jr $ra # Return from the function call
```

5 Instruction Formats

Instructions are represented as bits (just like numbers), so its a good idea to store instructions in memory just like data (why?). In MIPS Instruction Format, every instruction is represented as a fixed 32-bit word, and an instruction is further divided into different fields.

1) About MIPS Instruction Formats

(a) **I format**: used for instructions with immediates, lw and sw (since offset counts as an immediate), and branches (beq and bne).

- opcode: 6 bits, rs: 5 bits, rt: 5, immediate: 16 bits
- For branch: the immediate field is signed int and word-aligned.

(b) **J format**: used for general jumps, (j and jal). We may jump to anywhere in memory (why?).
• opcode: 6 bits, target address: 26 bits.
• New PC = \{(PC + 4)[31 ... 28], target address, 00\}

(c) **R format:** used for all other instructions.
• opcode: 6 bits, rs: 5 bits, rt: 5, rd: 5 bits, shamt: 5 bits, funct: 6 bits.
• the opcode field for all R-type instructions is 0.
6 Additional Practice - Writing MIPS Functions

Here is a general template for writing functions in 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