CS61C Summer 2016 Discussion 3 – MIPSII/Instruction Formats

1 Translating between C and MIPS

Translate between the C and MIPS code. You may want to use the MIPS Green Sheet as a reference. In all of the C examples, we show you how the different variables map to registers – you don't have to worry about the stack or any memory-related issues.

C	MIPS

`// Strcpy:`
`// $s1 -> char s1[]`
`// $s2 -> char *s2 =`
`//        malloc(sizeof(char)*7);`
`int i = 0;`
`do {`
`    s2[i] = s1[i];`
`    i++;`
`} while(s1[i] != '\0');`
`s2[i] = '\0';`
	`      addiu $t0, $0, 0`
	`Loop: addu $t1, $s1, $t0  # s1[i]`
	`      addu $t2, $s2, $t0  # s2[i]`
	`      lb   $t3, 0($t1)    # char is`
	`      sb   $t3, 0($t2)    # 1 byte!`
	`      addiu $t0, $t0, 1`
	`      addiu $t1, $t1, 1   # unnecessary line`
	`      lb    $t4, 0($t1)   # could use offset`
	`      bne   $t4, $0, Loop`
	`Done: sb   $t4, 1($t2)`

`// Nth_Fibonacci(n):`
`// $s0 -> n, $s1 -> fib`
`// $t0 -> i, $t1 -> j`
`// Assume fib, i, j are these values`
`int fib = 1, i = 1, j = 1;`
`if (n==0)      return 0;`
`else if (n==1) return 1;`
`n -= 2;`
`while (n != 0) {`
`    fib = i + j;`
`    j = i;`
`    i = fib;n--;`
`}`
`return fib;`
	`    ...`
	`    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: ...`

`// Collatz conjecture`
`// $s0 -> n`
`unsigned n;`
`L1: if (n % 2) goto L2;`
`goto L3;`
`L2: if (n == 1) goto L4;`
`n = 3 * n + 1;`
`goto L1;`
`L3: n = n >> 1;`
`goto L1;`
`L4: return n;`
	`L1:   addiu $t0, $0, 2`
	`      div $s0, $t0      # puts (n%2) in $hi`
	`      mfhi $t0          # sets $t0 = (n%2)`
	`      bne $t0, $0, L2`
	`      j L3`
	`L2:   addiu $t0, $0, 1`
	`      beq $s0, $t0, L4`
	`      addiu $t0, $0, 3`
	`      mul $s0, $s0, $t0`
	`      addiu $s0, $s0, 1`
	`      j L1`
	`L3:   srl $s0, $s0, 1`
	`      j L1`
	`L4:   ...`

2 MIPS Instruction Formats

Instructions are represented as bits (just like numbers), so it's 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 a 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 bits, 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 bits, rd: 5 bits, shamt: 5 bits, funct: 6 bits.
- the opcode field for all R-type instruction is 0 .

(2) Exercises:

(a) What instruction is 0x00008A03?

```
Hex -> bin:              0000 0000 0000 0000 1000 1010 0000 0011
0 opcode -> R-type:      000000 00000 00000 10001 01000 000011
sra $s1 $0 8
```

(b) What is the hexadecimal representation of instruction "addiu $a0, $0, 0xABC"

```
common mistake: 001001 00100 00000 0000ABC = 24800ABC = addiu $0, $a0, 0xABC
correct answer: 001001 00000 00100 0000ABC = 0x24040ABC
```

3 MIPS Addressing Modes

- 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 (**in word, so 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 (**"in word", with implicit 00 lowest bits**) to make a 32-bit address (used by J-formatinstructions)

(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.

```
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 were 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 problems assumption, we can use a jump. Assuming were jumping to label Foo

```
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 (youll 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___
```