Discussion #3: MIPS Introduction

Written by Justin Hsia (6/28/2011)

Note: I do not want to spend much time here going over the details. That's what your MIPS Green Card (http://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf) is for. That card has basically all the information you need on it, but it's up to you to 1) figure out how to read it properly and 2) know how to properly use that information.

Note: This set of discussion notes will focus on the assembly code side. We'll cover the machine language portion of MIPS later.

MIPS Registers:

There are 32 registers. You should be noticing that this number comes up a lot. You can reference each register either by its number or by its name (for example, $zero and $0 are the same). I group them by use below:

$zero: Useful everywhere! This little guy is pretty amazing and we will see the power of zero in all sorts of situations below.
$at, $k0-$k1: Reserved – do not use!
$v0-$v1, $a0-$a3, $ra: These are all used in function calls. We will see the details in the next set of discussion notes.
$fp, $sp, $gp: Pointers. Allow us to keep track of and access different parts of memory.
$s0-$s7, $t0-$t9: Registers used for holding variables and performing calculations.

MIPS Instruction Set:

The core instruction set can be found on the left column of the first page of the MIPS Green Card or P&H p.78. With the exception of the jumping instructions (and a few others), all of the instructions take three arguments. The arguments can be classified as follows (the exact nomenclature and abbreviations may vary slightly, but the gist is the same):

`src` – Source register. The contents of the register are read for use in the instruction.
`dst` – Destination register. Result of the instruction is stored here.
`imm` – Immediate (constant). You can write these integer values (can be negative) in decimal or hex.
`offset` – Address offset (constant). It's really just an `imm`, but used in the context of modifying an address.
`bAddr` – Base address (register). Used for accessing memory.
`brAddr` – Branch address (named). Not actually a name. Gets converted into an instruction offset based on where you define the branch name.

Learning the naming scheme for the instructions themselves is quite useful. Things like `i` = immediate, `u` = unsigned, `b` = branch or byte, `w` = word, and `j` = jump will help you remember both the purpose and syntax of each instruction.

MIPS Comments

Start with the '#' symbol and encompass the rest of the line. You will see this often.

Common C to MIPS Conversions:

Learning to convert C code to MIPS takes a bit of practice, but it is more formulaic than you might think. There is definitely some freedom in implementation, but in general we want to use fewer instructions and fewer registers/less memory. In the following examples, example C statements will be shown on the left and their MIPS equivalents will be shown on the right. I suggest keeping your MIPS Green Card nearby as you work through these. First try doing the conversion on your own and then compare against what's shown on the right. There are multiple ways of doing certain statements.

Note: Remember that MIPS does not use the C variable names. It is up to you to keep track of where you store your variables. In the following examples I will just use arbitrary registers, with a preference for stored registers for named variables and temporary registers for intermediate calculations.

Initialization/Assignment:

`1) int x;`	`1) # Nothing to do!`
`2) int y = 5;`	`2) addi $s1, $0, 5`
`3) z = y;`	`3) add $s2, $0, $s1`

Pointer Arithmetic: (essentially the same for arrays)

Assume the following initializations have already taken place:

```
int *ptri = (int *)malloc(100*sizeof(int)); \\ ptri stored in $s0
char *ptrc = (char *)malloc(100*sizeof(char)); \\ ptrc stored in $s1
```

`1) *ptrc = 5;`	`1) addi $t0, $0, 5   # constant`
	`   sw $t0, 0($s1)`

`2) *(ptrc + 4) = -20;`	`2) addi $t0, $0, -20 # constant`
	`   sw $t0, 4($s1)`

`3) *(ptri + 4) = 10;`	`3) addi $t0, $0, 10  # constant`
	`   addi $t1, $0, 4   # offset`
	`   sll $t1, $t1, 2   # int is 4 bytes`
	`   add $t1, $s0, $t1 # address`
	`   sw $t0, 0($t1)`

`4) int x = *(ptri + 3);`	`4) addi $t0, $0, 3   # offset`
	`   sll $t0, $t0, 2   # int is 4 bytes`
	`   add $t0, $s0, $t0 # address`
	`   lw $t1, 0($t0)    # x = $t1`

`5) int x = 2;`
`   int y = *(ptrc + x);`
	`5) addi $t0, $0, 2   # x = $t0`
	`   add $t1, $s1, $t0 # address`
	`   lb $t2, 0($t1)    # y = $t2`

You are responsible for keep track of the size of your data! Offsets in MIPS are in bytes, so each index of a `char` array moves the address by 1 whereas each index of an `int` array will typically move the address by 4. To get proper address offset, you can either use the `multi` instruction or `sll` (only works for powers of two).

The last example above (#5) is a little weird. Even the C code is "bad form" since you are indirectly casting a `char` into an `int` (no loss of data, though, so C shouldn't complain). ptrc is a char pointer, so the pointer arithmetic uses bytes instead of words. Then the `lb` instruction automatically sign-extends your loaded value, giving you the proper `int`.

Check: Convince yourself that `i*=pow(2,N)`, `i << N`, and `sll $t0, $t0, N` all accomplish the same task, assuming that `N` is a non-negative integer, `i>0`, and ignoring overflow cases.

Offsets are immediates, so unfortunately we can't use the value of a register directly. That's why we store the `bAddr+offset*sizeof(type)` in another register (such as `$t0`) and then access `0($t0)`. If you are dealing with a constant offset, then you can directly write in the correct offset (in example 2, we used 4 directly and in example 3, we could have replaced lines 2-5 with `sw $t0, 16($s0)`).

Control Flow – if & logical statements:

Hopefully the branch names `If`, `Else`, and `Done` used here are self-explanatory enough. Depending exactly on your logical statement, different organizations will be better than others:

Here assume `x` in `$s0`, `y` in `$s1`, and `z` in `$s2` are all integers.

`1) if(x == 5)`	
	```
	1)     addi $t0, $0, 5
	       beq $s0, $t0, If
	       # else instructions go here
	       j Done
	If:    # if instructions go here
	Done:  # end of if
	```
	OR alternatively:
	```
	       addi $t0, $0, 5
	       bne $s0, $t0, Else
	       # if instructions go here
	       j Done
	Else:  # else instructions go here
	Done:  # end of if
	```

`2) if(y > z)`	
	```
	2)     slt $t0, $s2, $s1
	       beq $t0, $0, Else
	       # if instructions go here
	       j Done
	Else:  # else instructions go here
	Done:  # end of if
	```

`3) if(z <= 5)`	
	```
	3)     addi $t1, $0, 5
	       slt $t0, $t1, $s2
	       bne $t0, $0, Else
	       # if instructions go here
	       j Done
	Else:  # else instructions go here
	Done:  # end of if
	```

As demonstrated in example 1, you can always swap `beq` and `bne` as long as you swap the locations of the `If` and `Else` branches (one is usually left unlabeled). With comparison operators (`>`, `>=`, `<`, `<=`), you have to be a little more careful with how you apply `slt` and `slti`. MIPS only has “set on less than” and no “set on less than or equal to”, so a statement like `x >= 6` is done by solving for `x < 6` instead, which is the logical opposite.

Control Flow – loops:

Again, assume `x` in `$s0`, `y` in `$s1`, and `z` in `$s2` are all integers.

`1) for(x = 0; x < z; x++) {`
`        /* some code */`
`   }`
 	
In general:
`   for(<init>; <test>; <step>) {`
`        /* some code */`
`   }`
	`1)     add $s0, $0, $0`
	`Loop: slt $t0, $s0, $s2`
	`      beq $t0, $0, Done`
	`      # for instructions`
	`      addi $s0, $s0, 1`
	`      j Loop`
	`Done: # end of for`
	
	In general:
	`      # <init> instruction(s)`
	`Loop: # <test> instruction(s) and`
	`             # branch to Done`
	`      # for instructions`
	`      # <step> instruction(s)`
	`      j Loop`
	`Done: # end of for`

`2) while(y >= 4) {`
`        /* some code */`
`   }`
	`2)`
	`Loop: slti $t0, $s1, 4`
	`      beq $t0, $0, Done`
	`      # while instructions`
	`      j Loop`
	`Done: # end of while`

`3) do {`
`        /* some code */`
`   } while(z != y);`
	`3)`
	`Loop: # do-while instructions`
	`      bne $s2, $s1, Loop`
	`      # end of do-while`

A do-while loop is simpler than a while loop, which is simpler than a for loop! Here the choice of `beq` vs. `bne` is fixed, since you want the loop code to follow immediately.

Check: Convert the following to MIPS:

```
switch(x) {
      case 1: x++; break;
      case 2: y += x;
      case 3: z = 4; break;
      default: x = 0;
}
```