CSE 30321 – Lecture 05-06 – In Class Example Handout

Part A: Discussion – RISC vs. CISC
- With Approach 2, more fetch and decodes (time overhead) but also simplified datpath
- With Approach 1, could need more functional units, could affect clock cycle time, etc.
- Discuss.

Part B: Discussion – Programmer Visibility & Procedure Calls
- Board discussion of register that is programmer visible (i.e. R1) and that which is not (i.e. PC, memory data register, etc.)
- To discuss:
  - o Are there multiple copies of function calls in assembly?
  - o How do they think this is handled?

Part C: Byte Addressing (1)

1 Kbit memory, 16 bit word addressable (i.e. can address data in words that are 16 bits…)
- (?) # of words: 1024 / 16 = 64 words
- (?) # of address bits: $2^6 = 64 \rightarrow$ 6 bits of address

Part D: Byte Addressing (2)
- Let's assume we have words/instructions/etc. that are 32 bits
- Let's also assume that we have 32-bits of address…

| 31 | | 24,23 | | 16,15 | | 8,7 | | 0 |
| l | Byte 1 | l | Byte 2 | l | Byte 3 | l | Byte 4 | l |

If we address to the byte, how many addresses do we need?

Answer: 4. Therefore, if instructions are 32 bits, and instructions and data share the same memory, and we want to go from one instruction to the next, need to do: PC ← PC + 4

Part E: Discussion & More Examples

Discussion: Course Goals / C++, Java Example

Look at: "while((j>0) && numbers(j) > index)"
                       j=j-1;
                       …

What if our processor was designed to implement instructions you learned about for the 6-instruction processor?
- Might do something like:
  - o MOV R1, #1   // Load constant 1 into R1
  - o SUB R2, R2, R1   // Assuming j = R2, use SUB

What if our processor was designed to implement instructions associated with MIPS?
- Could use a subtract immediate instruction (built into MIPS already)
  - o SUBi R2, R2, 1   // R2 = R2 – 1
- There are more bits to encode/do more "stuff"

Important take away: the concept here is the ISA – just like a for() or while() loop is a concept in programming. Different ISAs are like different programming languages – i.e. the syntax for a for() loop may be different in Matlab or C or Java – but you're still iterating through data or code in the same way.

Now, what about numbers(j -) ? (assume numbers is in memory)

- In MIPS, load instruction always assumes part of an address in a register
    - lw $2, 0($1)   // Says $2 ← Mem(0 + contents of $2)
- Note – offset does not have to be 0 either…
    - lw $2, 20($1)   // Says $2 ← Mem(20 + contents of $2)

- What about for 6-instruction processor?
    - Problem – unless we assume an instruction like you added in Problem 1 of your last HW, we can't index memory with a variable.
    - What if we present problem as below (assume there are 5 elements in "numbers"):
        - 0:   numbers(0)   10:   MOV R1, #1 ***
        - 1:   numbers(1)   11:   MOV R2, #5
        - 2:   numbers(2)   12:   MOV R3, d(5)
        - 3:   numbers(3)   13:   SUB R2, R2, R1
        - ….   ….
        -                            Can something go here to get numbers(4)?
        -                            ….
        -                            13+N: JMPZ R2, ***
    - Yes!
        - MOV R4, d(12)   // Load instruction encoding into R4
        - SUB R4, R4, R1   // Now, MOV R4, d(4) – change encoding itself
        - MOV d(12), R4   // Store encoding back into memory
    - This is equivalent to changing an instruction in a register in MIPS!

Question 1:
Example:
- Assume we want to perform the operation A[4] = h + A[12]
- Further assume that the constant *h* is stored in $17
- Also, the starting address of array A is stored in $18

Assuming that data words are 32 bits long (as is the case with MIPS), write a sequence of MIPS machine instructions to perform this operation.

==Ask class for solution.==

1st, need to remember MIPS words are byte addressable + aligned:

3	….	0
7	….	4
11	….	8
15	….	12

Implies: if we want A(12), we do *not* do: lw $2, 12($18)
Instead: need, starting address of A + (12th element x 4)

So, one way to get A(12):   lw $2, 48($18)   // $2 ← M(48 + R(18))

Another way: addi $3, $18, 48   // $3 ← R(18) + $18
                                lw $2, 0($3)   // $2 ← M(0 + R(3))

                                (note – 2 instructions instead of 1)

To finish: lw $2, 48($18)
                                add $3, $2, $17
                                sw ???, $3

??? = 16($18)
   - Note, *cannot* have lw $2, $4($6)
        o   lw = i-type therefore offset must be encoded in the instruction
   - (Works if known at compile time)
        o   If not, may need to use other method (a lot like pointers)

Question 2:
Given the swap procedure shown below, write a sequence of MIPS machine instructions to perform the operation.

```
void swap(int *y, int k)
{
        int tmp;
        tmp = v[k];
        v[k] = v[k+1];
        v[k+1] = tmp;
}
```

Assume that tmp = $8, starting address of v = $17, k = $18

Class question: What to first? Figure out address associated with V(k)

Here, may need to use register technique b/c *k is a variable*
   - k = array index, but we need k to be a multiple of 4

   - If k = 1, need 4; if k = 2, need 8 …
        o   Example: k   =   8   =   0001000   = 2^3 =   8
        o   Shift by 2: k   =   32   =   0100000   = 2^5 =   32

   - Multiply k by 4 – easiest way is to shift binary number left
        o   MIPS has instruction to do this…
                ▪   sll $20, $18, 2
        o   Gives us correct decimal value of k

   - Now, need to tack on starting address of array:
        o   Just use add instruction: add $20, $20, $17   // reuse $20

   - Now, $20 = physical address associated with v(k)

Class: finish rest of code…
        lw $8, 0($20)   // $8 = tmp (given in problem); get (v(k)
        lw $9, 4($20)   // read v(k+1)

```
        sw $9, 0($20)                    // swap in memory
        sw $8, 4($20)                    //
```

Question 3:
Write the sequence of MIPS instructions to accomplish the following:

```
        if (i == j)
                h = i + j;
        else
                h = i – j;
```

Hint #1:
  -   You shouldn't need more than 4 instructions
Hint #2:
  -   Think about using a bne or beq instruction
  -   (bne $t0, $t1, Label # if $t0 != $t1, goto Label)

Assume: i = $7   j = $8   k = $9

Answer:

```
        bne     $7, $8, else              // if $7 != $8, want to do "else"
        add     $9, $7, $8                // if $7 = $8, want to add $7 and $8 and store the result in $9
        jump    end if                    // if $7 = $8, want do nothing else
else:   sub     $9, $7, $8                // if $7 != $8, h = i – j
end if:
```

Question 4:
How would we implement a branch-if-less-than instruction?
  -   blt $s1, $2, Label

Hint:
  -   Use the set-on-less-than instruction
  -   slt $t0, $s1, $s2   # reg $t0 = 0 if $s1 >= $s2
                              # reg $t0 = 1 if $s1 < $s2

```
Answer:       slt     $1, $1, $2     // $1 reserved for assembler; might be reserved as "flag" bit
              bne     $1, $0, label  // $1 is set to 1 or 0 above, $0 = 0 by default
                                     // therefore, if s1 < s2, $1 = 1
                                     // then 1 !=0, so branch
```