EXERCISES #1

One of the problems in using VMT for virtual memory is the size of VMT (i.e., large memory space needed for holding VMT). One of the solutions for this problem is “inverted page table”.

The concept of the inverted page table (IPT) is that for each page in the physical memory (instead of for each page in the virtual memory) a virtual page number is specified (the dirty flag can be included in IPT), if any virtual memory page is currently associated to the physical memory page (otherwise some invalid virtual page number, such as “-1” is specified).

Questions:

(1) What is the primary advantage in using IPT for implementing virtual memory?

(2) What is the primary disadvantage in using IPT for implementing virtual memory?
EXERCISE #2

“Loop unrolling (A.K.A. “loop-iteration merging”)” is a static performance optimization. Loop unrolling tries to eliminate especially RAW data dependency by merging multiple loop iterations into one iteration, when a loop structure repeats for a specific number of times in a program source code. Thus, loop unrolling is applied to “for” loops (because the number of the iterations in each for loop structure is known in the compile time), while loop unrolling is NOT applicable to “while” loops (because the number of the iterations is usually not known in the compile time). Assume the following loop-structure using “for”:

```plaintext
for (i = 0; i < 120; i++)
{
    array_01[i] = array_01[i] + 10;
    array_02[i] = array_02[i] - 50;
}
```

which is translated into the following assembly instructions:

```plaintext
array_01: .word u0, u1, u2, u3, .... u119   # 120 data elements
array_02: .word v0, v1, v2, v3, ... v119   # 120 data elements
main:
    la  $a0, array_01                # $a1 is the pointer to “array_02”
    la  $a1, array_02                # $a1 is the pointer to “array_02”
    li  $s0, 120                     # $s0 is the loop counter
BEGIN_LOOP:
    lw  $t0, ($a0)                  # load array_01[x] to $t0 register
    add $t0, $t0, 10                # add a constant “10” to $t0 register
    sw  ($a0), $t0                  # save $t0 register to array_01[x]
    add $a0, $a0, 4                 # advance the pointer to the next element
    lw  $t0, ($a1)                  # load array_02[x] to $t0 register
    sub $t0, $t0, 50                # subtract a constant “50” from $t0 register
    sw  ($a1), $t0                  # save $t0 register to array_02[x]
    add $a1, $a1, 4                 # advance the pointer to the next element
    sub $s0, $s0, 1                 # decrease the loop counter by one
    bne $s0, 0, BEGIN_LOOP          # repeat if the loop counter is not zero
END_LOOP:  some instruction

Assumptions:

(a) The processor is a five-stage pipeline processors
(b) “R” is performed in ID stage and “W” is performed in WB stage
(c) Registers $a0 and $a1 are pointer, which points to the beginning of “array_01” and “array_02” respectively.
(d) Each element in “array_01” and “array_02” is four bytes (see the figure in the next page)
(e) This processor has infinite number of $t$ registers

**Question:** For the following for loop structure, (a) find how much can the for-loop be faster when loop unrolling (in terms of the number of the processor cycles from the label “BEGIN_LOOP” to “END_LOOP”)? (b) Explain how the improvement is achieved (it is suggested that you explain “how” by showing the assembly instructions after loop-unrolling has been applied). For this question, 80% of the credit is for (b).

Show all your work (in such a way that Dr. Fujinoki can understand your idea(s)).

**Hint #1:** load (lw) and store (sw) instructions allow “offset” such as “lw $t0, 12($a0)”.

**Hint #2:** the structure of the two arrays (“array_01” and “array_02”):