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?

The primary advantage: the very small amount of information is needed to implement virtual memory.

For the following system:
- 16GB of virtual memory address
• 64MB of the physical memory
• 4KB page size

The number of the virtual memory pages: \(16\text{GB/4KB} = (2^4 \times 2^{30})/(2^2 \times 2^{10}) = 2^2 \times 2^{20}\)
The number of the physical memory pages: \(64\text{MB/4KB} = (2^6 \times 2^{20})/(2^2 \times 2^{10}) = 2^4 \times 2^{10}\)

Each element in IPT needs: \(\log_2(2^2 \times 2^{20}) = 22\) bits for each of \((2^4 \times 2^{10})\) physical memory pages:

\[
(2^4 \times 2^{10}) \times 22 \text{ bits} = 16\text{K} \times 22 \text{ bits} = 352\text{K} \text{ bits} = 44\text{KB}
\]

Note: the VMT for the same system requires 8MB of space to hold VMT.

**The primary disadvantage**: slow (takes long time to find the physical memory page - the memory manager has to scan the “virtual memory page number) for finding the physical memory page that holds the target virtual memory page.
**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    # $a1 is the pointer to “array_02”
lw ($a0), $t0       # 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
li $s0, 120         # $s0 is the loop counter
BEGIN_LOOP:

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”):
Suggested Solution:

As we discussed in the classroom, a 5-stage pipeline processor will cause 3-cycle stall for each RAW dependency:

<table>
<thead>
<tr>
<th>move</th>
<th>$t1, $t2:</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>ME</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$t3, ($t1) 5:</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
</tbody>
</table>

Pipeline stalls

Without the code optimization (i.e., without loop unrolling):

```
1  lw  $t0, ($a0)    # load array_01[x] to $t0 register
2  STALL
3  STALL
4  STALL
5  add  $t0, $t0, 10  # add a constant “10” to $t0 register
6  STALL
7  STALL
8  STALL
9  sw  ($a0), $t0    # save $t0 register to array_01[x]
10 add  $a0, $a0, 4   # advance the pointer to the next element
11 lw  $t0, ($a1)    # load array_02[x] to $t0 register
12 STALL
13 STALL
14 STALL
15 sub  $t0, $t0, 50  # subtract a constant “50” from $t0 register
16 STALL
17 STALL
18 STALL
19 sw  ($a1), $t0    # save $t0 register to array_02[x]
20 add  $a1, $a1, 4   # advance the pointer to the next element
24 sub  $s0, $s0, 1   # decrease the loop counter by one
25 STALL
26 STALL
27 STALL
28 bne  $s0, 0, BEGIN_LOOP  # repeat if the loop counter is not zero
```

The number of the processor cycles needed: \(120 \times 28 = 3360\) cycles
With loop-unrolling:

Three more rounds of the same loop structure should be merged to eliminate the three stall cycles for each RAW dependency.

```
1  sub  $s0, $s0, 4    # decrease the loop counter by four
1  lw  $t0, ($a0)    # load array_01[x] to $t0 register
2  lw  $t1, 4($a0)   # load array_01[x+1] to $t1 register
3  lw  $t2, 8($a0)   # load array_01[x+2] to $t2 register
4  lw  $t3, 12($a0)  # load array_01[x+3] to $t3 register
5  add $t0, $t0, 10  # add a constant “10” to $t0 register
6  add $t1, $t1, 10  # add a constant “10” to $t1 register
7  add $t2, $t2, 10  # add a constant “10” to $t2 register
8  add $t3, $t3, 10  # add a constant “10” to $t3 register
9  sw  ($a0), $t0    # save $t0 register to array_01[x]
10 sw  4($a0), $t1   # save $t1 register to array_01[x+1]
11 sw  8($a0), $t2   # save $t2 register to array_01[x+2]
12 sw  12($a0), $t3  # save $t3 register to array_01[x+3]
13 lw  $t0, ($a1)    # load array_02[x] to $t0 register
14 lw  $t1, 4($a1)   # load array_02[x+1] to $t1 register
15 lw  $t2, 8($a1)   # load array_02[x+2] to $t2 register
16 lw  $t3, 12($a1)  # load array_02[x+3] to $t3 register
17 sub $t0, $t0, 50   # subtract a constant “50” from $t0 register
18 sub $t1, $t1, 50   # subtract a constant “50” from $t1 register
19 sub $t2, $t2, 50   # subtract a constant “50” from $t2 register
20 sub $t3, $t3, 50   # subtract a constant “50” from $t3 register
21 sw  ($a1), $t0    # save $t0 register to array_02[x]
22 sw  4($a1), $t1   # save $t1 register to array_02[x+1]
23 sw  8($a1), $t2   # save $t2 register to array_02[x+2]
24 sw  12($a1), $t3  # save $t3 register to array_02[x+3]
25 add $a0, $a0, 16   # advance the pointer by 16 bytes
26 add $a1, $a1, 16   # advance the pointer by 16 bytes
27 bne $s0, 0, BEGIN_LOOP  # repeat if the loop counter is not zero
```

The number of the processor cycles needed: \((120/4) \times 27\) = 810

By applying loop-unrolling as shown above, we achieve: \(3360/810 = 4.18\) times faster