1 Vector Processing [15 points]

Consider the following piece of code:

\[
\text{for (i = 0; i < 100; i ++)} \\
\quad A[i] = ((B[i] \times C[i]) + D[i])/2;
\]

(a) Translate this code into assembly language using the following instructions in the ISA (note the number of cycles each instruction takes is shown next to each instruction):

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Number of Cycles</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>Ri, X</td>
<td>1</td>
<td>Ri ← address of X</td>
</tr>
<tr>
<td>LD</td>
<td>Ri, Rj, Rk</td>
<td>11</td>
<td>Ri ← MEM[Rj + Rk]</td>
</tr>
<tr>
<td>ST</td>
<td>Ri, Rj, Rk</td>
<td>11</td>
<td>MEM[Rj + Rk] ← Ri</td>
</tr>
<tr>
<td>MOVI</td>
<td>Ri, Imm</td>
<td>1</td>
<td>Ri ← Imm</td>
</tr>
<tr>
<td>MUL</td>
<td>Ri, Rj, Rk</td>
<td>6</td>
<td>Ri ← Rj x Rk</td>
</tr>
<tr>
<td>ADD</td>
<td>Ri, Rj, Rk</td>
<td>4</td>
<td>Ri ← Rj + Rk</td>
</tr>
<tr>
<td>ADD</td>
<td>Ri, Rj, Imm</td>
<td>4</td>
<td>Ri ← Rj + Imm</td>
</tr>
<tr>
<td>RSHFA</td>
<td>Ri, Rj, amount</td>
<td>1</td>
<td>Ri ← RSHFA (Rj, amount)</td>
</tr>
<tr>
<td>BRcc</td>
<td>X</td>
<td>1</td>
<td>Branch to X based on condition codes</td>
</tr>
</tbody>
</table>

Assume one memory location is required to store each element of the array. Also assume that there are 8 registers (R0 to R7).

Condition codes are set after the execution of an arithmetic instruction. You can assume typically available condition codes such as zero, positive, and negative.

Solution:

```assembly
MOVI R1, 99 // 1 cycle
LEA R0, A // 1 cycle
LEA R2, B // 1 cycle
LEA R3, C // 1 cycle
LEA R4, D // 1 cycle

LOOP:
LD R5, R2, R1 // 11 cycles
LD R6, R3, R1 // 11 cycles
MUL R7, R5, R6 // 6 cycles
LD R5, R4, R1 // 11 cycles
ADD R8, R7, R5 // 4 cycles
RSHFA R9, R8, 1 // 1 cycle
ST R9, R0, R1 // 11 cycles
ADD R1, R1, ~1 // 4 cycles
BRGEZ R1 LOOP // 1 cycle
```

How many cycles does it take to execute the program?

Solution:

\[5 + 100 \times 60 = 6005\]
(b) Now write Cray-like vector assembly code to perform this operation in the shortest time possible. Assume that there are 8 vector registers and the length of each vector register is 64. Use the following instructions in the vector ISA:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Number of Cycles</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>Vst, #n</td>
<td>1</td>
<td>Vst ← n (Vst = Vector Stride Register)</td>
</tr>
<tr>
<td>LD</td>
<td>Vln, #n</td>
<td>1</td>
<td>Vln ← n (Vln = Vector Length Register)</td>
</tr>
<tr>
<td>VLD</td>
<td>Vi, X</td>
<td>11, pipelined</td>
<td></td>
</tr>
<tr>
<td>VST</td>
<td>Vi, X</td>
<td>11, pipelined</td>
<td></td>
</tr>
<tr>
<td>Vmul</td>
<td>Vi, Vj, Vk</td>
<td>6, pipelined</td>
<td></td>
</tr>
<tr>
<td>Vadd</td>
<td>Vi, Vj, Vk</td>
<td>4, pipelined</td>
<td></td>
</tr>
<tr>
<td>Vrshfa</td>
<td>Vi, Vj, amount</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Solution:

LD     Vln, 50
LD     Vst, 1
VLD    V1, B
VLD    V2, C
Vmul   V4, V1, V2
VLD    V3, D
Vadd   V6, V4, V3
Vrshfa V7, V6, 1
VST    V7, A
VLD    V1, B + 50
VLD    V2, C + 50
Vmul   V4, V1, V2
VLD    V3, D + 50
Vadd   V6, V4, V3
Vrshfa V7, V6, 1
VST    V7, A + 50

How many cycles does it take to execute the program on the following processors? Assume that memory is 16-way interleaved.

(i) Vector processor without chaining, 1 port to memory (1 load or store per cycle)

Solution:

The third load (VLD) can be pipelined with the add (VADD). However as there is just only one port to memory and no chaining, other operations cannot be pipelined.

Processing the first 50 elements takes 346 cycles as below

| 1 | 1 | 11 | 49 | 1 | 49 | 6 | 49 |
| 11 | 49 | 4 | 49 | 1 | 49 | 11 | 49 |

Processing the next 50 elements takes 344 cycles as shown below (no need to initialize Vln and Vst as they stay at the same value).

| 11 | 49 | 11 | 49 | 6 | 49 |
| 11 | 49 | 4 | 49 | 1 | 49 | 11 | 49 |

Therefore, the total number of cycles to execute the program = 690 cycles
(ii) Vector processor with chaining, 1 port to memory

**Solution:**

In this case, the first two loads cannot be pipelined as there is only one port to memory and the third load has to wait until the second load has completed. However, the machine supports chaining, so all other operations can be pipelined.

Processing the first 50 elements takes 242 cycles as below:

```
| 1 | 1 | 11 | 49 | 11 | 49 |
| 6 | 49 |
| 11 | 49 |
| 4 | 49 |
| 1 | 49 |
| 11 | 49 |
```

Processing the next 50 elements takes 240 cycles (same time line as above, but without the first 2 instructions to initialize VIn and Vst).

Therefore, the total number of cycles to execute the program = 482 cycles

(iii) Vector processor with chaining, 2 read ports and 1 write port to memory

**Solution:**

Assuming an in-order pipeline.

The first two loads can also be pipelined as there are two ports to memory. The third load has to wait until the first two loads complete. However, the two loads for the second 50 elements can proceed in parallel with the store.

```
| 1 | 1 | 11 | 49 |
| 1 | 11 | 49 |
| 6 | 49 |
| 11 | 49 |
| 4 | 49 |
| 1 | 49 |
| 11 | 49 |
| 11 | 49 |
| 11 | 49 |
| 6 | 49 |
| 11 | 49 |
| 4 | 49 |
| 1 | 49 |
| 11 | 49 |
```

Therefore, the total number of cycles to execute the program = 215 cycles

2 More Vector Processing (Optional) [0 points]

You are studying a program that runs on a vector computer with the following latencies for various instructions:

- VLD and VST: 50 cycles for each vector element; fully interleaved and pipelined.
- VADD: 4 cycles for each vector element (fully pipelined).
- VMUL: 16 cycles for each vector element (fully pipelined).
- VDIV: 32 cycles for each vector element (fully pipelined).
- VRSHF (right shift): 1 cycle for each vector element (fully pipelined).
Assume that:

- The machine has an in-order pipeline.
- The machine supports chaining between vector functional units.
- In order to support 1-cycle memory access after the first element in a vector, the machine interleaves vector elements across memory banks. All vectors are stored in memory with the first element mapped to bank 0, the second element mapped to bank 1, and so on.
- Each memory bank has an 8 KB row buffer.
- Vector elements are 64 bits in size.
- Each memory bank has two ports (so that two loads/stores can be active simultaneously), and there are two load/store functional units available.

(a) What is the minimum power-of-two number of banks required in order for memory accesses to never stall? (Assume a vector stride of 1.)

**Solution:**

64 banks (because memory latency is 50 cycles and the next power of two is 64)

(b) The machine (with as many banks as you found in part a) executes the following program (assume that the vector stride is set to 1):

```
VLD V1 ← A
VLD V2 ← B
VADD V3 ← V1, V2
VMUL V4 ← V3, V1
VRSHF V5 ← V4, 2
```

It takes 111 cycles to execute this program. What is the vector length?

**Solution:**

```
VLD |-----50-----|---(VLEN-1)---|
VLD |1|-----50-----|---(VLEN-1)---|
VADD |--4--|---(VLEN-1)---|
VMUL |---16---|---(VLEN-1)---|
VRSHF |1|---(VLEN-1)---|
```

1 + 50 + 4 + 16 + 1 + (VLEN - 1) = 71 + VLEN = 111 → VLEN = 40 elements

If the machine did not support chaining (but could still pipeline independent operations), how many cycles would be required to execute the same program?

**Solution:**

```
VLD |-----50-----|---(VLEN-1)---|
VLD |1|-----50-----|---(VLEN-1)---|
VADD |--4--|---(VLEN-1)---|
VMUL |---16---|---(VLEN-1)---|
VRSHF |1|---(VLEN-1)---|
```

50 + 1 + 4 + 16 + 1 + 4 * (VLEN - 1) = 68 + 4 * VLEN = 228 cycles
(c) The architect of this machine decides that she needs to cut costs in the machine’s memory system. She reduces the number of banks by a factor of 2 from the number of banks you found in part (a) above. Because loads and stores might stall due to bank contention, an arbiter is added to each bank so that pending loads from the oldest instruction are serviced first. How many cycles does the program take to execute on the machine with this reduced-cost memory system (but with chaining)?

**Solution:**

```
VLD [0] |----50----| bank 0 (takes port 0)
       ... [31] |--31--|----50----| bank 31
       [32] |---50---| bank 0 (takes port 0)
       ... [39] |---7--| bank 7
VLD [0] |1|----50----| bank 0 (takes port 1)
       ... [31] |1|--31--|----50----| bank 31
       [32] |---50----| bank 0 (takes port 1)
       ... [39] |---7--| bank 7
VADD |--4--| (tracking last elements)
VMUL |--16--|
VRSHF |1|
```

\[(B[39]: 1 + 50 + 50 + 7) + 4 + 16 + 1 = 129 \text{ cycles}\]

Now, the architect reduces cost further by reducing the number of memory banks (to a lower power of 2). The program executes in 279 cycles. How many banks are in the system?

**Solution:**

```
VLD [0] |----50----|
       ... [8] |---50----|
       ... [16] |---50--|
       ... [24] |---50--|
       ... [32] |---50--|
       ... [39] |---7--|
VLD [39] |1|
VADD |--4--|
VMUL |---16--|
VRSHF |1|
```

\[5\times 50 + 7 + 1 + 4 + 16 + 1 = 279 \text{ cycles} \rightarrow 8 \text{ banks}\]

(d) Another architect is now designing the second generation of this vector computer. He wants to build a multicore machine in which 4 vector processors share the same memory system. He scales up the number of banks by 4 in order to match the memory system bandwidth to the new demand. However, when he simulates this new machine design with a separate vector program running on every core, he finds that the average execution time is longer than if each individual program ran on the original single-core system with 1/4 the banks. Why could this be? Provide concrete reason(s).
Solution:
Row-buffer conflicts (all cores interleave their vectors across all banks).

What change could this architect make to the system in order to alleviate this problem (in less than 20 words), while only changing the shared memory hierarchy?

Solution:
Partition the memory mappings, or using better memory scheduling.

3  VLIW [15 points]
You are using a tool that transforms machine code that is written for the MIPS ISA to code in a VLIW ISA. The VLIW ISA is identical to MIPS except that multiple instructions can be grouped together into one VLIW instruction. Up to $N$ MIPS instructions can be grouped together ($N$ is the machine width, which depends on the particular machine). The transformation tool can reorder MIPS instructions to fill VLIW instructions, as long as loads and stores are not reordered relative to each other (however, independent loads and stores can be placed in the same VLIW instruction). You give the tool the following MIPS program (we have numbered the instructions for reference below):

(01) lw $t0 ← 0($a0)
(02) lw $t2 ← 8($a0)
(03) lw $t1 ← 4($a0)
(04) add $t6 ← $t0, $t1
(05) lw $t3 ← 12($a0)
(06) sub $t7 ← $t1, $t2
(07) lw $t4 ← 16($a0)
(08) lw $t5 ← 20($a0)
(09) srlv $s2 ← $t6, $t7
(10) sub $s1 ← $t4, $t5
(11) add $s0 ← $t3, $t4
(12) sllv $s4 ← $t7, $s1
(13) srlv $s3 ← $t6, $s0
(14) sllv $s5 ← $s0, $s1
(15) add $s6 ← $s3, $s4
(16) add $s7 ← $s4, $s6
(17) srlv $t0 ← $s6, $s7
(18) srlv $t1 ← $t0, $s7

(a) Draw the dataflow graph of the program Represent instructions as numbered nodes (01 through 18), and flow dependences as directed edges (arrows).
(b) When you run the tool with its settings targeted for a particular VLIW machine, you find that the resulting VLIW code has 9 VLIW instructions. What minimum value of $N$ must the target VLIW machine have?

**Solution:**

$N = 3$ (see VLIW program below). If $N = 2$, then the VLIW program must have at least 11 MIPS instructions, and the number of VLIW instructions either stays the same or decreases as width is increased by one MIPS instruction.

(c) Write the MIPS instruction numbers (from the code above) corresponding to each VLIW instruction, for this value of $N$. When there is more than one MIPS instruction that could be placed into a VLIW instruction, choose the instruction that comes earliest in the original MIPS program.

**Solution:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW Instruction 1: 01 02 03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 2: 04 05 06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 3: 07 08 09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 4: 10 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 5: 12 13 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 6: 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 7: 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 8: 17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 9: 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) You find that the code is still not fast enough when it runs on the VLIW machine, so you contact the VLIW machine vendor to buy a machine with a larger machine width $N$. What minimum value of $N$
would yield the maximum possible performance (i.e., the fewest VLIW instructions), assuming that all MIPS instructions (and thus VLIW instructions) complete with the same fixed latency and assuming no cache misses?

Solution:

\( N = 6 \). This is the maximum width of the dataflow graph and results in 7 VLIW instructions (see below). If \( N = 5 \), then the VLIW program will instead have 8 VLIW instructions. Increasing \( N \) further does not allow any more MIPS instructions to be parallelized in wider VLIW instructions.

e) Write the MIPS instruction numbers corresponding to each VLIW instruction, for this optimal value of \( N \). Again, as in part (c) above, pack instructions such that when more than one instruction can be placed in a given VLIW instruction, the instruction that comes first in the original MIPS code is chosen.

Solution:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW Instruction 1:</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>05</td>
<td>07</td>
<td>08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 2:</td>
<td>04</td>
<td>06</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 3:</td>
<td>09</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 4:</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 5:</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 6:</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 7:</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 8:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW Instruction 9:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(f) A competing processor design company builds an in-order superscalar processor with the same machine width \( N \) as the width you found in part (b) above. The machine has the same clock frequency as the VLIW processor. When you run the original MIPS program on this machine, you find that it executes slower than the corresponding VLIW program on the VLIW machine in part (b). Why could this be the case?

Solution:

Concurrently fetched instructions can be dependent in a superscalar processor, requiring bubbles in the pipeline to be processed. A VLIW code translator can reorder instructions to minimize such bubbles.

Note that the superscalar processor is in-order in this question.

g) When you run some other program on this superscalar machine, you find it runs faster than the corresponding VLIW program on the VLIW machine. Why could this be the case?

Solution:

VLIW code must have explicit NOPs; the superscalar processor does not require these NOPs. Higher code density results in a higher I-cache hit rate and lower required fetch bandwidth.

4 Virtual Memory [15 points]

An ISA supports an 8-bit, byte-addressable virtual address space. The corresponding physical memory has only 128 bytes. Each page contains 16 bytes. A simple, one-level translation scheme is used and the page table resides in physical memory. The initial contents of the frames of physical memory are shown below.
A three-entry translation lookaside buffer that uses Least Recently-Used (LRU) replacement is added to this system. Initially, this TLB contains the entries for pages 0, 2, and 13. For the following sequence of references, put a circle around those that generate a TLB hit and put a rectangle around those that generate a page fault. What is the hit rate of the TLB for this sequence of references? (Note: LRU policy is used to select pages for replacement in physical memory.)

References (to pages): 0, 13, 5, 2, 14, 14, 13, 6, 6, 13, 15, 14, 15, 13, 4, 3.

**Solution:**
References (to pages): (0), (13), 5, 2, [14], (14), 13, [6], (6), (13), [15], 14, (15), (13), [4], [3].
TLB Hit Rate = 7/16

(a) At the end of this sequence, what three entries are contained in the TLB?

**Solution:**
4, 13, 3

(b) What are the contents of the 8 physical frames?

**Solution:**
Pages 14, 13, 3, 2, 6, 4, 15, Page table

5  **Page Table Bits [5 points]**

(a) What is the purpose of the “reference” or “accessed” bit in a page table entry?

**Solution:**
To aid page replacement.

(b) Describe what you would do if you did not have a reference bit in the PTE. Justify your reasoning and/or design choice.

**Solution:**
Pick a random page to replace when a page fault occurs.

(c) What is the purpose of the dirty or modified bit in a page table entry?

**Solution:**
To enable writeback of only dirty pages (rather than all pages) to disk.

(d) Describe what you would do if you did not have a modified bit in the PTE. Justify your reasoning and/or design choice.

**Solution:**
Write back all pages to disk.

Alternative answer: the OS could map all pages read-only by default. On a page-fault due to a write, if the program has permission to change the page, the operating system remaps the page as read-write and also knows that the page has become dirty.
6 Caching [15 points]

Below, we have given you four different sequences of addresses generated by a program running on a processor with a data cache. Cache hit ratio for each sequence is also shown below. Assuming that the cache is initially empty at the beginning of each sequence, find out the following parameters of the processor’s data cache:

- Associativity (1, 2 or 4 ways)
- Block size (1, 2, 4, 8, 16, or 32 bytes)
- Total cache size (256 B, or 512 B)
- Replacement policy (LRU or FIFO)

Assumptions: all memory accesses are one byte accesses. All addresses are byte addresses.

<table>
<thead>
<tr>
<th>Sequence No.</th>
<th>Address Sequence</th>
<th>Hit Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0, 2, 4, 8, 16, 32</td>
<td>0.33</td>
</tr>
<tr>
<td>2</td>
<td>0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0</td>
<td>0.33</td>
</tr>
<tr>
<td>3</td>
<td>0, 64, 128, 256, 512, 256, 128, 64, 0</td>
<td>0.33</td>
</tr>
<tr>
<td>4</td>
<td>0, 512, 1024, 0, 1536, 0, 2048, 512</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Solution:

**Cache block size - 8 bytes**

For sequence 1, only 2 out of the 6 accesses (specifically those to addresses 2 and 4) can hit in the cache, as the hit ratio is 0.33. With any other cache block size but 8 bytes, the hit ratio is either smaller or larger than 0.33. Therefore, the cache block size is 8 bytes.

**Associativity - 4**

For sequence 2, blocks 0, 512, 1024 and 1536 are the only ones that are reused and could potentially result in cache hits when they are accessed the second time. Three of these four blocks should hit in the cache when accessed for the second time to give a hit rate of 0.33 (3/9).

Given that the block size is 8 and for either cache size (256B or 512B), all of these blocks map to set 0. Hence, an associativity of 1 or 2 would cause at most one or two of these four blocks to be present in the cache when they are accessed for the second time, resulting in a maximum possible hit rate of less than 3/9. However, the hit rate for this sequence is 3/9. Therefore, an associativity of 4 is the only one that could potentially give a hit rate of 0.33 (3/9).

**Total cache size - 256 B**

For sequence 3, a total cache size of 512 B will give a hit rate of 4/9 with a 4-way associative cache and 8 byte blocks regardless of the replacement policy, which is higher than 0.33. Therefore, the total cache size is 256 bytes.

**Replacement policy - LRU**

For the aforementioned cache parameters, all cache lines in sequence 4 map to set 0. If a FIFO replacement policy were used, the hit ratio would be 3/8, whereas if an LRU replacement policy were used, the hit ratio would be 1/4. Therefore, the replacement policy is LRU.