1 Potpourri

(a) Full Pipeline

Keeping a processor pipeline full with useful instructions is critical for achieving high performance. What are the three fundamental reasons why a processor pipeline cannot always be kept full?

Reason 1.

Reason 2.

Reason 3.

(b) Out-of-Order vs. Dataflow

When does an instruction execute in a dataflow processor?

When does the fetch of an instruction happen in an out-of-order execution processor?

What structure holds the instructions until they are ready to execute in an out-of-order processor?

(c) Tomasulo’s Algorithm

Here is the state of the reservation stations in a processor during a particular cycle (\(\times\) denotes an unknown value):
## ADD Reservation Station

<table>
<thead>
<tr>
<th>Tag</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>D</td>
<td>×</td>
<td>1</td>
<td>×</td>
<td>27</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>×</td>
<td>3</td>
<td>0</td>
<td>E</td>
<td>×</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>B</td>
<td>×</td>
<td>0</td>
<td>A</td>
<td>×</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

## MUL Reservation Station

<table>
<thead>
<tr>
<th>Tag</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0</td>
<td>B</td>
<td>×</td>
<td>0</td>
<td>C</td>
<td>×</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>×</td>
<td>16</td>
<td>0</td>
<td>B</td>
<td>×</td>
</tr>
</tbody>
</table>

What is wrong with this picture?
2 Out-of-Order Execution

In this problem, we will give you the state of the Register Alias Table (RAT) and Reservation Stations (RS) for a Tomasulo-like out-of-order execution engine. Your job is to determine the original sequence of five instructions in program order.

The out-of-order machine in this problem behaves as follows:

- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine dispatches one instruction per cycle into the reservation stations, in program order. Dispatch occurs during the decode stage.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.
- When a value is captured (at a reservation station) or written back (to a register) in this machine, the old tag that was previously at that location is not cleared; only the valid bit is set.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- Both the adder and multiplier are fully pipelined. Add instructions take 2 cycles. Multiply instructions take 4 cycles.
- When an instruction completes execution, it broadcasts its result, and dependent instructions can begin execution in the next cycle if they have all operands available.
- When multiple instructions are ready to execute at a functional unit, the oldest ready instruction is chosen.

Initially, the machine is empty. Five instructions then are fetched, decoded, and dispatched into reservation stations, before any instruction executes. Then, one instruction completes execution. Here is the state of the machine at this point, after the single instruction completes:

<table>
<thead>
<tr>
<th>Rat</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>R2</td>
<td>0</td>
<td>A 37</td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>X 500</td>
</tr>
<tr>
<td>R4</td>
<td>0</td>
<td>Y 255</td>
</tr>
<tr>
<td>R5</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>R6</td>
<td>0</td>
<td>Z 73</td>
</tr>
<tr>
<td>R7</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src 1</th>
<th>Tag</th>
<th>Value</th>
<th>Src 2</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td>1 500</td>
<td></td>
<td>Y</td>
<td>0 -</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
<td>1 20</td>
<td></td>
<td>-</td>
<td>1 17</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src 1</th>
<th>Tag</th>
<th>Value</th>
<th>Src 2</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>1 50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 37</td>
<td></td>
<td>A</td>
<td>0 -</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>0 -</td>
</tr>
</tbody>
</table>

Initially, the machine is empty. Five instructions then are fetched, decoded, and dispatched into reservation stations, before any instruction executes. Then, one instruction completes execution. Here is the state of the machine at this point, after the single instruction completes:
(a) Give the five instructions that have been dispatched into the machine, in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: “opcode destination ⇐ source1, source2.”

(b) Now assume that the machine flushes all instructions out of the pipeline and restarts execution from the first instruction in the sequence above. Show the full pipeline timing diagram below for the sequence of five instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fifth instruction.

As we saw in class, use “F” for fetch, “D” for decode, “E1,” “E2,” “E3,” and “E4” to signify the first, second, third and fourth cycles of execution for an instruction (as required by the type of instruction), and “W” to signify writeback. You may or may not need all columns shown.

Finally, show the state of the RAT and reservation stations after 8 cycles in the blank figures below.
3 The GPU Strikes Back!

We define the *SIMD utilization* of a program run on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program.

The following code segment is run on a GPU. Each thread executes a *single iteration* of the shown loop. Assume that the data values of the arrays A, B, and C are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 5 instructions in each thread.) A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU.

```c
for (i = 0; i < (512*1024); i++) {
    if (B[i] < 0) {
        A[i] = A[i] * C[i];
        B[i] = A[i] + B[i];
        C[i] = C[i] + 1;
        B[i] = B[i] + 1;
    }
}
```

(a) How many warps does it take to execute this program?

(b) When we measure the SIMD utilization for this program with one input set, we find that it is 9/40. What can you say about arrays A, B, and C? Be precise.

A: 

B: 

C: 

(c) Is it possible for this program to yield a SIMD utilization of 20% (circle one)?

YES           NO
If YES, what should be true about arrays A, B, and C for the SIMD utilization to be 20%? Be precise.

A: 
B: 
C: 

If NO, explain why not.

(d) Is it possible for this program to yield a SIMD utilization of 100% (circle one)?

YES  NO

If YES, what should be true about arrays A, B, C for the SIMD utilization to be 100%? Be precise.

A: 
B: 
C: 

If NO, explain why not.
4 Microcoded Machines

Microcoding is a powerful technique to perform complex computations on a simple datapath. In this problem, you will sketch out a microcode-controlled single-bus implementation of the MIPS R2000 ISA.

In a single-bus processor, the primary interconnect between state elements and functional units is a shared bus. In each cycle, only one unit can drive data on the bus, while all other units may listen and selectively latch the data. Below is the starting point of a single-bus microarchitecture.

```
Assume that there is a microcode control ROM that can drive all of the control signals shown based on a current state and specify a next state. As an example, below is a possible microcode sequence for the three-register ADD instruction. Assume that other states which are not shown handle instruction fetch and PC update. Thus, at the initial state shown here, the instruction register already contains the instruction.
```

```
State | nextState | lePC | enPC | leIR | enIR | leA | leB | ALU | ALU_OP | leIDX | RF | Mem | RF | Mem | comment
-----|-----------|-----|------|-----|------|-----|-----|-----|--------|------|----|-----|----|-----|--------
ADD_x | ADD_x+1   | 0   | 0    | 0   | 0    | 0   | 0   | 0   | 0      | 0    | 0 | 0    | 0 | 0    | Latch rs into IDX
ADD_x | ADD_x+2   | 0   | 0    | 0   | 0    | 0   | 0   | 0   | 0      | 0    | 0 | 0    | 0 | 0    | Read RF[rs] into A
ADD_x | ADD_x+3   | 0   | 0    | 0   | 0    | 0   | 0   | 0   | 0      | 0    | 0 | 0    | 0 | 0    | Read RF[rt] into B
ADD_x | ADD_x+4   | 0   | 0    | 0   | 0    | 0   | 0   | 0   | 0      | 0    | 0 | 0    | 0 | 0    | Read RF[rd] into IDX
ADD_x | ADD_x+5   | 0   | 0    | 0   | 0    | 0   | 0   | 0   | 0      | 0    | 0 | 0    | 0 | 0    | A+B into RF[rd]
```

For this homework problem, you will implement a new instruction, ADDM, in microcode. ADDM performs an addition where one operand is loaded from memory and the other operand comes from a register, and the result is stored back into a register. It is an R-type instruction with the following semantics:

```
RF[rd] = M[RF[rs]] + RF[rt]
```

In other words, the instruction loads the word in memory at the address specified by register rs, then adds the loaded value to the value in register rt, and stores the result in register rd. Write out a microcode sequence like the one in the table above for the ADDM instruction. You can assume that when accessing memory (with the MEM_R signal asserted), the microcode sequencer will stall until the memory provides the data.
5 Out-of-Order Execution

Out-of-order processors make efficient use of their functional units by executing instructions according to the flow of data between them.

(a) In class, we learned about a graphical way of showing the data dependencies (edges) of instructions (vertices) using a data flow graph. For the instruction stream below, draw the corresponding data flow graph.

\[
\begin{align*}
\text{ADD } r3 &\leftarrow r1, r2 \\
\text{MUL } r4 &\leftarrow r1, r3 \\
\text{ADD } r5 &\leftarrow r8, r9 \\
\text{DIV } r6 &\leftarrow r1, r3 \\
\text{ADD } r7 &\leftarrow r6, r5 \\
\end{align*}
\]

(b) Of course, while a data flow graph is a helpful way of visualizing what goes on in an out-of-order processor, real machines execute instructions using multiple pipeline stages. Take the code from above and find the number of cycles it would take to execute on a processor with in-order fetch and out-of-order dispatch. Assume the following:

- It takes one cycle to decode an instruction and another cycle to issue the decoded instruction into a reservation station. These two stages can be pipelined.
- There are separate functional units for ADD, MUL and DIV, and each functional unit has its own reservation station.
- Tags are broadcast in the same cycle their corresponding operation finishes executing.
- A single tag broadcast bus exists and if more than one reservation station entry to the same execution unit is ready to be dispatched, the older reservation station entry will use the bus and the younger ones will have to stall. Similarly, if more than one instruction contends for writing to the ROB or architectural register file during a single cycle, the oldest one receives access and the younger ones must stall.
- ADDs take 3 cycles and are pipelined, MULs take 5 cycles and are pipelined, and DIVs take 7 cycles and are pipelined.

How many cycles will the program take to execute on an in-order fetch and out-of-order dispatch machine, under the assumptions above?

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) How many cycles will the program take to execute on an in-order fetch and in-order dispatch machine, under the same assumptions as above?

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8/14
6 Vector Processing

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, etc.
- Each memory bank has an 8KB 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.)

(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?

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

(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)?
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?
7 Tracing the Cache

Assume you have three toy CPUs: 6808-D, 6808-T, and 6808-F. All three CPUs feature one level of cache. The cache size is 128 bytes, the cache block size is 32 bytes, and the cache uses LRU replacement. The only difference between the three CPUs is the associativity of the cache:

- 6808-D uses a direct mapped cache.
- 6808-T uses a two-way associative cache.
- 6808-F uses a fully associative cache.

You run the SPECMem3000 program to evaluate the CPUs. This benchmark program tests only memory read performance by issuing read requests to the cache. Assume that the cache is empty before you run the benchmark. The cache accesses generated by the program are as follows, in order of access from left to right:


Each letter represents a unique cache block. All 8 cache blocks are contiguous in memory. However, the ordering of the letters does not necessarily correspond to the ordering of the cache blocks in memory. For 6808-D, you observe the following cache misses in order of generation:


(a) By using the above trace, please identify which cache blocks are in the same set for the 6808-D processor. Please be clear.

(b) Please write down the sequence of cache misses for the 6808-F processor in their order of generation. (Hint: You might want to write down the cache state after each request).

(c) For 6808-T, you observed the following five cache misses in order of generation:

A, B, H, G, E

But, unfortunately, your evaluation setup broke before you could observe all cache misses for the 6808-T. Using the given information, which cache blocks are in the same set for the 6808-T processor?

(d) Please write down the sequence of cache misses for the 6808-T processor in their order of generation.

(e) What is the cache miss rate for each processor?

6808-D:
8 Programming a Systolic Array

Figure 1 shows a systolic array processing element.

Each processing element takes in two inputs, M and N, and outputs P and Q. Each processing element also contains an “accumulator” R that can be read from and written to. The initial value of the “accumulator” is 0.

Figure 2 shows a systolic array composed of 9 processing elements. The smaller boxes are the inputs to the systolic array and the larger boxes are the processing elements. You will program this systolic array to perform the following calculation:

\[
\begin{bmatrix}
c_{00} & c_{01} & c_{02} \\
c_{10} & c_{11} & c_{12} \\
c_{20} & c_{21} & c_{22}
\end{bmatrix}
= 
\begin{bmatrix}
a_{00} & a_{01} & a_{02} \\
a_{10} & a_{11} & a_{12} \\
a_{20} & a_{21} & a_{22}
\end{bmatrix}
\times 
\begin{bmatrix}
b_{00} & b_{01} & b_{02} \\
b_{10} & b_{11} & b_{12} \\
b_{20} & b_{21} & b_{22}
\end{bmatrix}
\]

In each time cycle, each processing element will take in its two inputs, perform any necessary actions, and write on its outputs. The time cycle labels on the input boxes determine which time cycle the inputs will be fed into their corresponding processing elements. Any processing element input that is not driven will default to 0, and any processing element that has no output arrow will have its output ignored.

After all the calculations finish, each processing element’s “accumulator” will hold one element of the final result matrix, arranged in the correct order.

(a) Please describe the operations that each individual processing element performs, using mathematical equations and the variables M, N, P, Q and R.

Figure 1: A systolic array processing element

P =

Q =

R =

(b) Please fill in all 30 input boxes in Figure 2 so that the systolic array computes the correct matrix multiplication result described on the previous page. (Hint: Use \(a_{ij}\) and \(b_{ij}\).)
Figure 2: A systolic array