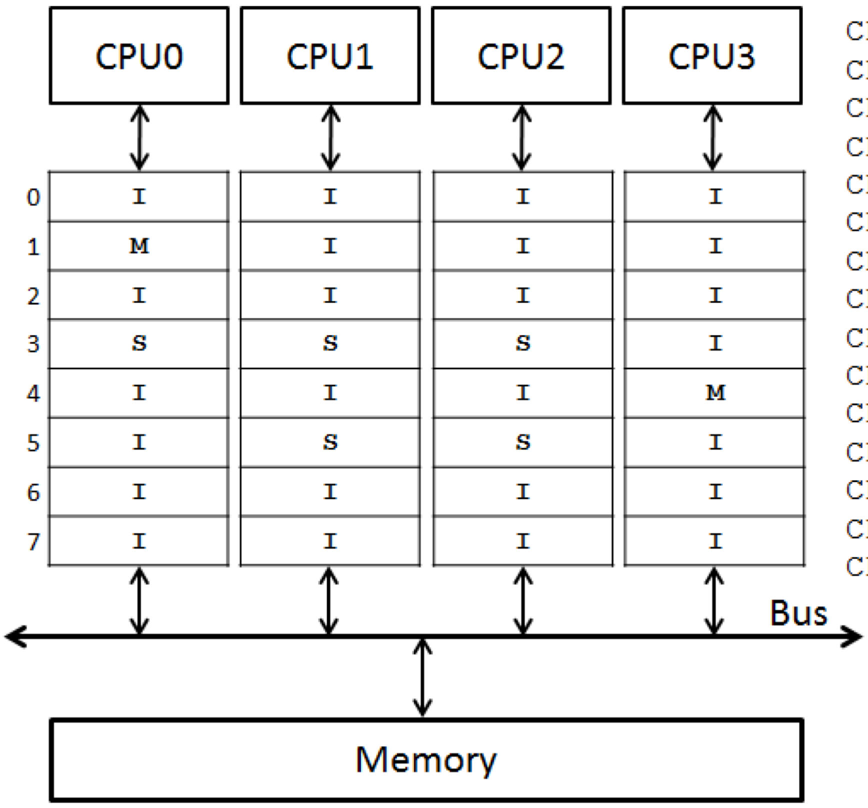
Cache Coherence

In this problem, we will examine the MSI, MESI, and MOESI protocols for maintaining coherence among caches in a multiprocessor system. The system we will examine contains 4 processors, each of which contains a direct mapped cache with 8 cache lines. At some point in execution, the coherence state of this memory system is as seen below. Cache requests are then made to the caches from their respective CPUs as shown on the right.
Assume all requests within a particular cache index are for the same line, such that we have no conflict misses or evictions.



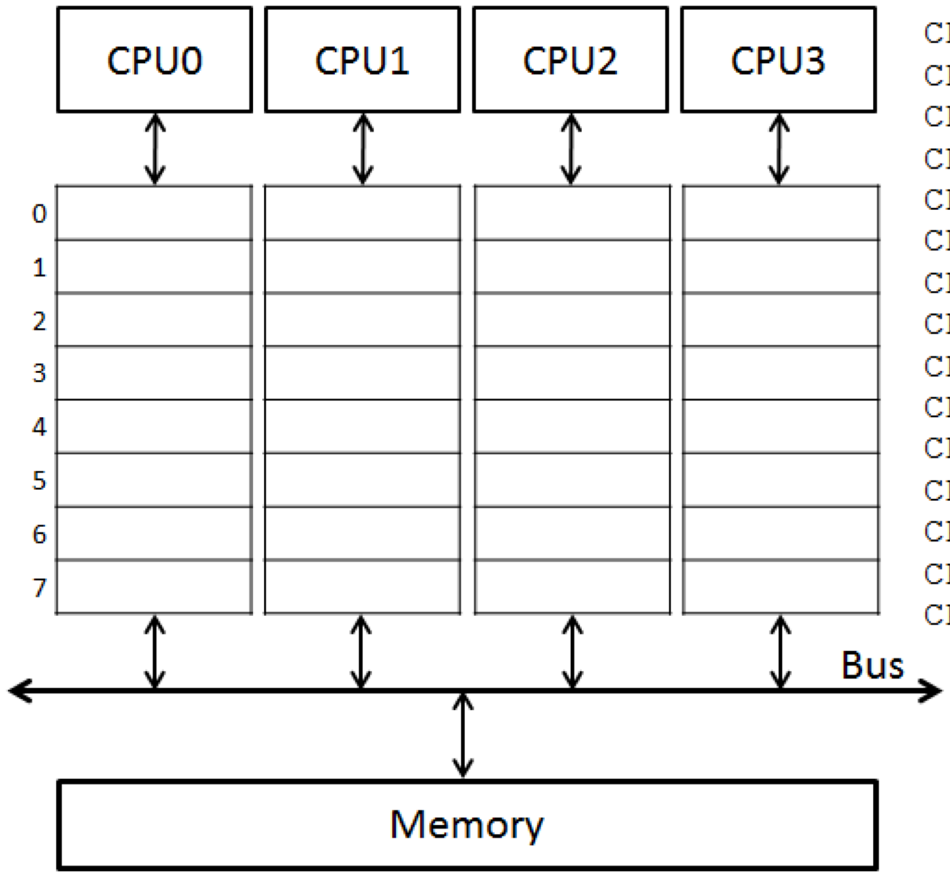
	CPU0	CPU1	CPU2	CPU3
0	I	I	I	I
1	M	I	I	I
2	I	I	I	I
3	S	S	S	I
4	I	I	I	M
5	I	S	S	I
6	I	I	I	I
7	I	I	I	I

```
CPU 0 - Read  0
CPU 2 - Write 1
CPU 3 - Read  2
CPU 1 - Write 3
CPU 0 - Read  4
CPU 3 - Write 5
CPU 2 - Write 6
CPU 1 - Read  7
CPU 0 - Read  0
CPU 2 - Read  2
CPU 1 - Read  7
CPU 3 - Read  6
CPU 0 - Read  0
CPU 1 - Read  1
CPU 2 - Read  0
```

a) MSI Protocol (X pts)

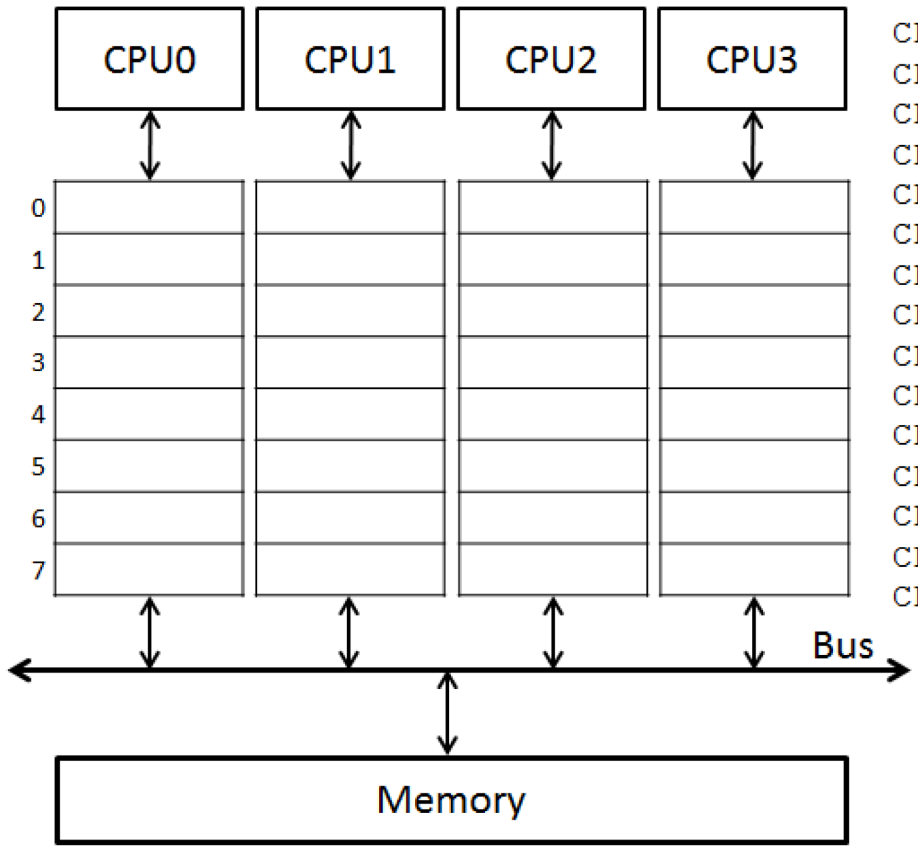
Show the final state of the system using the MSI protocol for cache coherence. A state diagram for the MSI protocol is provided in Figure X, near the back of the exam.

i) M = Modified, read/write state
ii) S = Shared, read only state
iii) I = Invalid



```
CPU 0 - Read  0
CPU 2 - Write 1
CPU 3 - Read  2
CPU 1 - Write 3
CPU 0 - Read  4
CPU 3 - Write 5
CPU 2 - Write 6
CPU 1 - Read  7
CPU 0 - Read  0
CPU 2 - Read  2
CPU 1 - Read  7
CPU 3 - Read  6
CPU 0 - Read  0
CPU 1 - Read  1
CPU 2 - Read  0
```

b) MESI Protocol (X pts)
Show the final state of the system using the MESI (Illinois) protocol for cache coherence. A state diagram for the MSI protocol is provided in Figure X, near the back of the exam.
i) M = Modified, read/write state
ii) E = Exclusive, read-only state, no other processor holds line
iii) S = Shared, read only state, other processors hold line
iv) I = Invalid

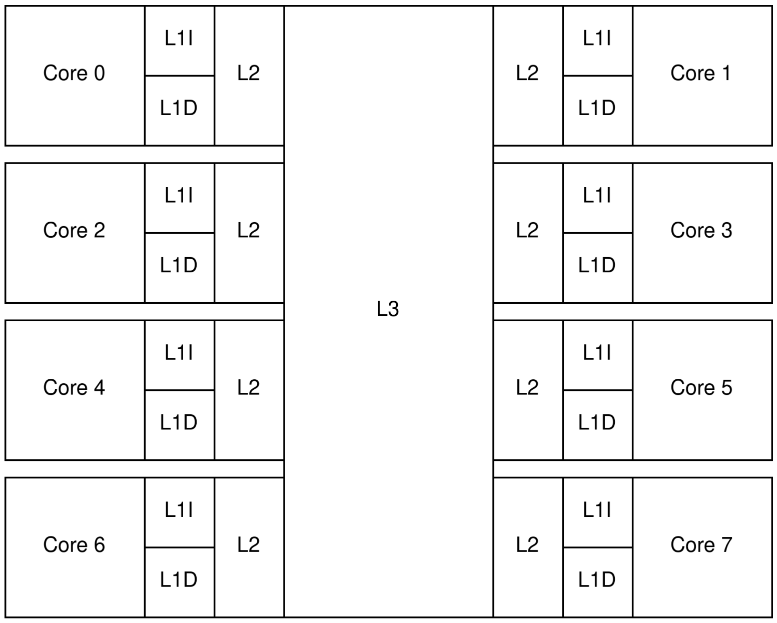


```
CPU 0 - Read  0
CPU 2 - Write 1
CPU 3 - Read  2
CPU 1 - Write 3
CPU 0 - Read  4
CPU 3 - Write 5
CPU 2 - Write 6
CPU 1 - Read  7
CPU 0 - Read  0
CPU 2 - Read  2
CPU 1 - Read  7
CPU 3 - Read  6
CPU 0 - Read  0
CPU 1 - Read  1
CPU 2 - Read  0
```

c) How many bus transactions are saved when using the MESI protocol versus the MSI protocol? (X pts)

d) What is the benefit of an Owned state in a cache coherence protocol? When would you make the transition into the Owned?(X pts)

3. a) This is a multicore design inspired by a commercially manufactured processor. It has split L1I, L1D caches per core, a unified L2 cache per core and an L3 cache shared across all 8 cores. You can assume a physically addressed system. This following diagram illustrates one "core cluster" of this design.



i) An inclusive L2 cache will include all values present in the L1I and L1D caches. How does this increase or decrease the latency of a snooping based cache coherence protocol? You do not need to consider the effects related to changes in cache hit rates.
Explain in up to 2 sentences.

ii) Self-modifying code involves the instructions being used as data in the program. The program changes the data at locations where instructions are present and therefore changes instructions. The code snippet provided is an example. Depending on whether r2 and r3 are equal, TAKE_ACTION could be replaced by a nop, so instructions themselves are modified based on the program execution.

```
                # instructions before
                beq   r2, r3, TAKE_ACTION
                add   r4, r0, 0x13 # 0x13 is nop
                la    r5, TAKE_ACTION # load address into r5
                store r4, 0(r5) # Replace with nop
TAKE_ACTION:
                add   r1, r1, 1
                # other instructions
```

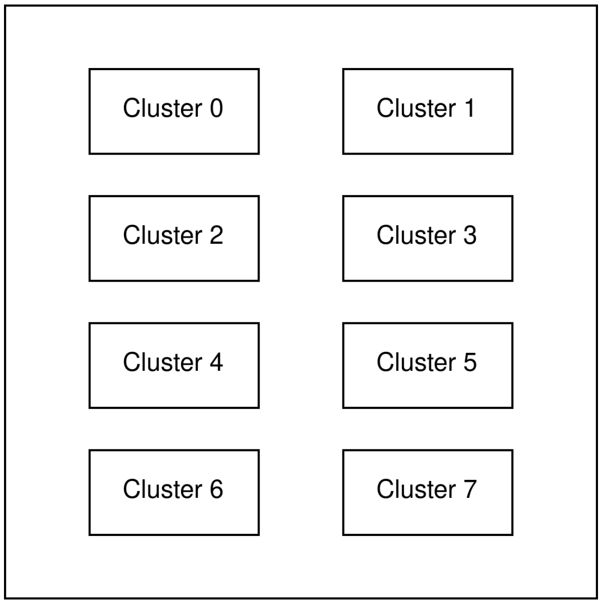
Assume the L2 cache is not inclusive and the L1 caches need to be kept coherent with the Illinois protocol. How can we use fewer states in the L1 instruction cache such that the L1 instruction cache still works with caches employing MESI? Explain your answer for full credit.

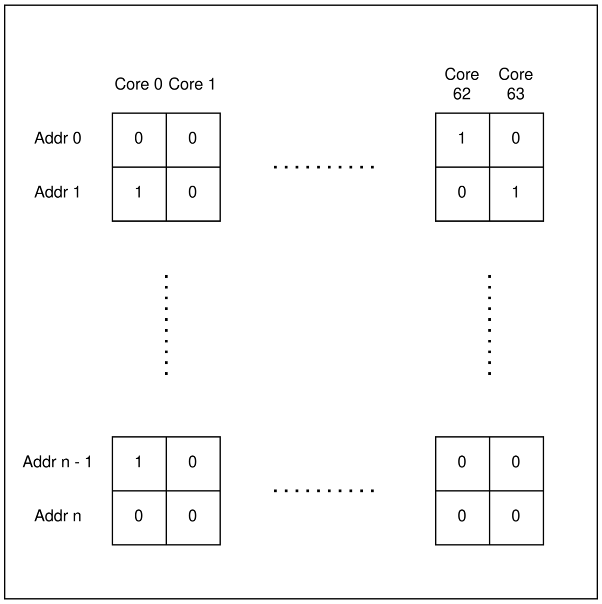
If we use the MOESI protocol instead, what states should we take out from the instruction cache? Explain your answer for full credit.

b) We implement a 64-core processor with 8 of these clusters shown previously. Let this be the setup for the rest of the question. Let each cluster store up to 32MB (2^15) of unique cachelines. L1<->L2 and L2<->L3 are kept coherent with snooping, but L3 caches are kept coherent using directory-based cache coherence. A single directory is employed.

The directory uses a full-bit vector representation illustrated below. Only the presence bits are shown. The row is indexed using memory addresses, with each row uniquely identifying all addresses in 1 cacheline. Every row has a presence bit for each core. A "1" implies the cacheline is currently present in the corresponding core's cache.

Each cacheline has a size of 64 bytes (2^6).





a. How many bits of space does the directory's 'presence bits' take up in total, if the total physical address space is 1TB (2^40)?

b. What is the maximum number of presence bits in the directory that can be 1 at any given instance in time? In other words, at any point in time, how many squares can have a value of 1?

c. In 1 sentence, give a reason why the values in a and b differ (very little / significantly) ?

d. Other than using a single presence bit for a cluster rather than for a core, suggest a way to reduce the space used by the directory.

Memory Consistency

1. You've learned about Dekker's algorithm in class; we've provided Wikipedia's implementation of Dekker's algorithm below in pseudocode. The main intention is to maintain mutual exclusion in the critical section, meaning that both threads should not be able to enter the critical section simultaneously. Now, consider **Processor Consistency**; in PC the same guarantees as Sequential consistency exist, except loads are allowed to bypass stores (as long as they honor local dependencies). The rules for PC are given below:

    a. Before a **LOAD** is allowed to perform w.r.t. any processor, all previous **LOAD** accesses must be performed w.r.t. every processor **(notice no mention of store in the latter clause)**

    b. Before a **STORE** is allowed to perform w.r.t. any processor all previous **LOAD**/**STORE** accesses must be performed w.r.t. every processor

    As an example, consider the following codes:

P0:	P1:
A = 1	B = 1
print(B)	print(A)

    While under sequential consistency this code is not allowed to print 00, under processor consistency the loads can bypass the stores, resulting in 00 being a legal outcome. Below is an example interleaving of events under processor consistency where this can happen legal under processor consistency:

    P0 stores 1 to A but the store is **not yet visible to P1** (one possible reason why is due to some temporary store buffer holding the write until the memory system can process it)
    P0 allows the load to B before its store to A is visible to everyone (including P1), and thus **reads and prints 0**
    P1 stores 1 to B and it **immediately is visible to A**
    P1 loads A, **reads 0**, and prints it (since P0's store is not yet visible)
    P0's store to A is **eventually visible to P1** at some point in the future

    Take a look at Dekker's algorithm below and determine whether it can still maintain mutual exclusivity under Processor Consistency. If it cannot, give a legal interleaving of events under Processor Consistency where both threads end up in the critical section. You may refer to the code by line number and thread e.g. p1.11 refers to when p1 is on line 11 of the code.

```
  variables
        wants_to_enter : array of 2 booleans
        turn : integer

    wants_to_enter[0] ← false
    wants_to_enter[1] ← false
    turn ← 0   // or 1
```

```
p0:
     wants_to_enter[0] ← true
     while wants_to_enter[1] {
        if turn ≠ 0 {
           wants_to_enter[0] ←
false
           while turn ≠ 0 {
             // busy wait
           }
           wants_to_enter[0] ← 1
true
        }
     }

     // critical section
     ...
     turn ← 1
     wants_to_enter[0] ← false
     // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ←
false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ←
true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

VLIW

A VLIW machine has the following characteristics:

1 load/store unit. A load has a latency of 2 cycles but is fully pipelined, and there are no cache misses
2 integer ALU/Branch units. single cycle.

Every instruction goes through a fetch, decode, execute+memory and writeback stage. All forwarding paths are present. Some instructions may take more than one cycle in the execute+memory stage.

Note that all sub instructions are executed in parallel, and no forwarding is allowed within an instruction. Writing to the same register multiple times in the same instruction is disallowed

The ISA for our VLIW machine resembles RISC V, except that there can be up to 3 instructions on each line separated by semicolons. Here is an example.

```
addi x1, x1, 4; lw x2, 0(x4); add x3, x6, x7
```

Consider the following code sequence:

```
for (i=0; i<256; i++)
{
        if (A[i] > 0) C[i] = A[i] + B[i];
}
```

The corresponding RISC-V ASM is:
Assume the following initial values:
x1 = A (Address of array)
x2 = B (Address of array)
X3 = C (Address of array)
X4 = 0

```
loop:
1. lw x5, 0(x1)                    // x5 = A[i]
2. bleq x5, x0, skip               // if (A[i]<=0) goto skip
3. lw x6, 0(x2)               // x6 = B[i]
4. add x7, x5, x6                  // x7 = x5 + x6
5. sw x7, 0(x3)                    // C[i] = x7

skip:
6. addi x1, x1, 4                  //i++
7. addi x2, x2, 4
8. addi x3, x3, 4
9. addi x4, x4, 1
10. slti x8, x4, 256
11. bnez x8, loop                  if (i < 256) loop
```

We ask you to re-write the code above for our VLIW machine. Write NOP for any subinstruction you cannot fill.

Minimize the total number of instructions used. (Trivial answers with only one useful sub instruction per instruction will not receive any credit)

INT1	INT2	LD/ST

Accelerator

Ben Bitdiddle has just started working at Big Matrix™ where all they do is matrix multiplications. Turns out, this simple operation is used across many fields, like Scientific Computing, Machine Learning and Quantum Computing.

For reference, matrix multiplication refers to the following operation, where r1, r2, and r3 are rows and c1, c2 and c3 are columns.

$$\begin{array}{ccccc}
 & & \begin{array}{ccc} \vec{c_1} & \vec{c_2} & \vec{c_3} \\ \downarrow & \downarrow & \downarrow \end{array} & & \\
\begin{array}{c} \vec{r_1} \rightarrow \\ \vec{r_2} \rightarrow \\ \vec{r_3} \rightarrow \end{array} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} & \bullet & \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} & = & \begin{bmatrix} \vec{r_1} \bullet \vec{c_1} & \vec{r_1} \bullet \vec{c_2} & \vec{r_1} \bullet \vec{c_3} \\ \vec{r_2} \bullet \vec{c_1} & \vec{r_2} \bullet \vec{c_2} & \vec{r_2} \bullet \vec{c_3} \\ \vec{r_3} \bullet \vec{c_1} & \vec{r_3} \bullet \vec{c_2} & \vec{r_3} \bullet \vec{c_3} \end{bmatrix}
\end{array}$$

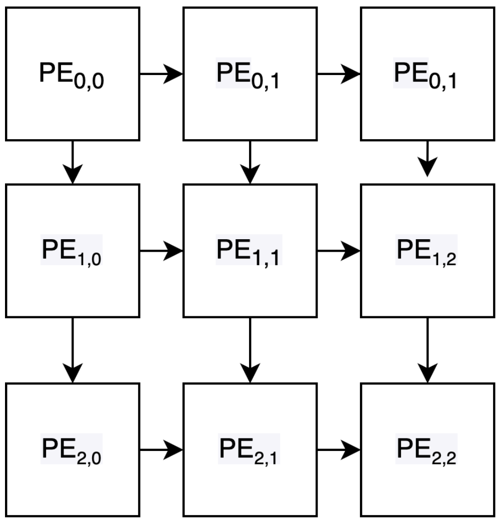
Big Matrix™ was using general purpose RISC-V cores to do all their matrix multiplications.

Ben found this inefficient and wanted to build accelerators for matrix multiplications. He wants your help to convince his manager.

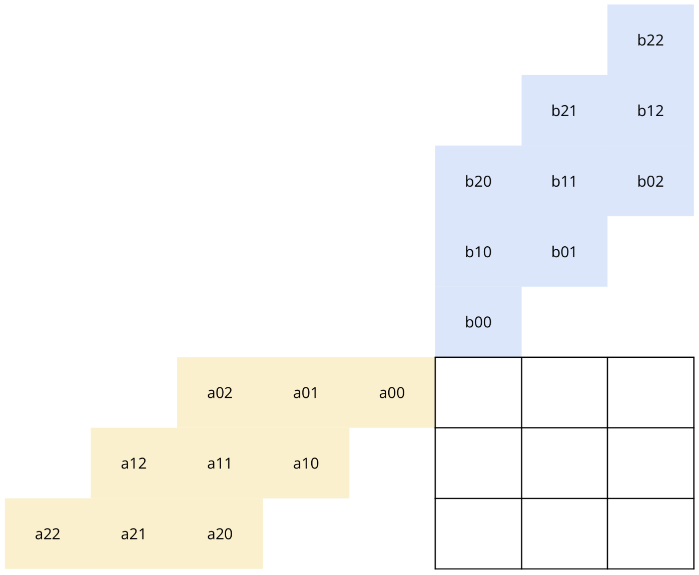
1. Why are accelerators more appropriate for this workload compared to general purpose processors?

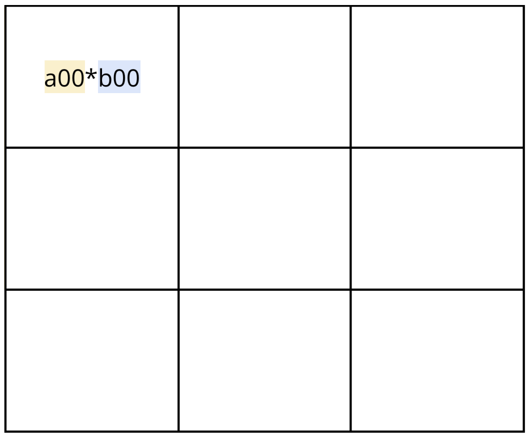
2. With your convincing answer, Ben was able to convince his manager that they need an Accelerator. Ben was asked to lead the project, and decided to build an accelerator for 3 x 3 matrix multiplication.

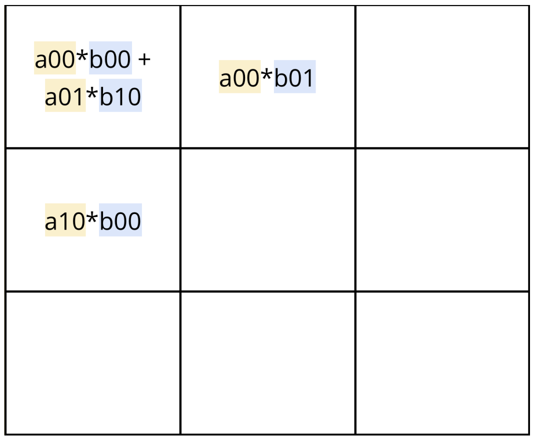
   Ben proposes a systolic array architecture.(Explain Systolic Arrays?) The architecture is 3 x 3 grid of processing element, each element capable of performing a Multiply and Accumulate (C = A*B + C) operation in one cycle.



The matrix multiplication is mapped to the systolic array in the following manner:







How many cycles will this accelerator take to compute a 3x3 matrix?

3. All the input matrices are generated by the RISC-V core and exist in the last level cache of the core, and need to be transferred to the accelerator before computation can occur. Assuming that it takes 100 cycles to transfer one element to the Accelerator’s on-chip memory, how long will it take to transfer both the inputs for the 3x3 matrix multiply?

4. When it is time to demo Ben’s new accelerator, he realizes that his proposed design is slower than the RISC-V core. What could be the reason?

5. After benchmarking the software implementation, Ben finds that it takes n^3(where n is the number of elements), cycles on the RISC-V core to perform the matrix multiplication. He asks for your help to redesign this accelerator so that it can be faster than the general purpose RISC-V core. Ben also tells you that most matrices at Big Matrix™ are much bigger than 3x3.

POWER

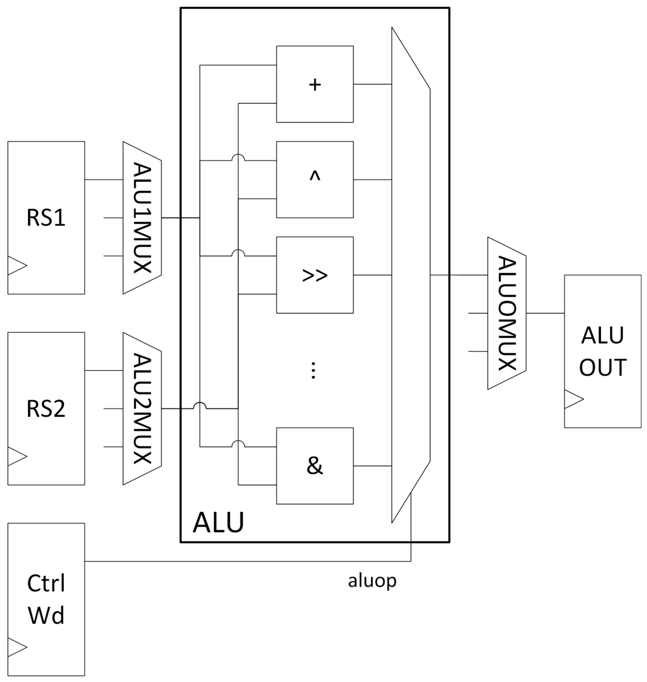
1. Recall that this is the provided ALU module in MP2, and likely the one you used in MP4 as well:

```
module alu
import rv32i_types::*;
(
    input alu_ops aluop,
    input [31:0] a, b,
    output logic [31:0] f
);

always_comb
begin
    unique case (aluop)
        alu_add:  f = a + b;
        alu_sll:  f = a << b[4:0];
        alu_sra:  f = $signed(a) >>> b[4:0];
        alu_sub:  f = a - b;
        alu_xor:  f = a ^ b;
        alu_srl:  f = a >> b[4:0];
        alu_or:   f = a | b;
        alu_and:  f = a & b;
    endcase
end

endmodule : alu
```

   You might have noticed that this implementation of ALU is just a glorified MUX whose inputs are the output of 8 different operations on the same operands. Regardless of the desired computation, all 8 operations are being performed at the same time, but only one is chosen to move forward. Here is a quick illustration of what this code would look like inside the pipeline (some signals are omitted for simplicity):



a. Suppose that you absolutely do not want your CPU to do unnecessary work because it is one of the valuable friends you made along the way. Draw a schematic of the revised ALU which only performs the necessary operation for that cycle. Include any registers (or latch if you like), multiplexers and decoders you want to add. Also show any other elements that you find necessary to include just like the illustration above. Your new design does not necessarily need to have the same interface as shown above, you are also free to move things across pipeline stages, but the functionality should stay the same. Draw a box around what is now considered the new ALU module.
b. One reason you might want to avoid unnecessary work is to save energy/power. Discuss the circumstance/condition/parameter when you design will: ("never" is a possible answer)
   i. Have higher peak power.
   ii. Have lower peak power.
   iii. Use more energy.
   iv. Use less energy.

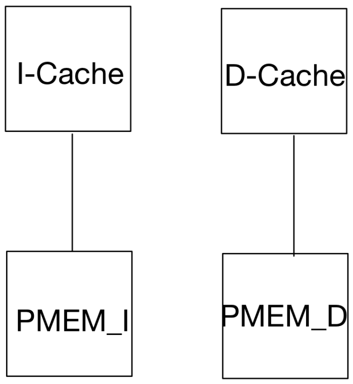
2. Ben Bitdiddle and Betty Bitdiddle are both pro gamers. They both overclock their computer to get the optimal gaming experience. They both try to get their CPU and GPU frequency as high as possible. However, they disagree slightly on how to tune the voltage.
   a. Betty says that she should increase the core voltage while overclocking. What might be her rationale? (Hint: think about the IV curve of a MOSFET and the threshold voltage)
   b. Ben insists that he should not increase the voltage while overclocking. What might be his rationale? (Hint: think about the limiting factor of the system and how

reducing the voltage helps.) Is there some pitfall Ben will fall into if he overclocks his CPU this way?

MP4

Fun fact:"Ying" means "win" in Mandarin.

1. In your MP4 design, you may notice the arbiter hurts the overall performance since it cannot serve instruction and data requests concurrently. CanYing is motivated by it and came up with an idea that to have separate memory modules for instructions and data:



a. Which cache should be larger & which physical memory should have higher capacity on a data intensive program?

b. Even for this data intensive program, with good spatial locality, there were many cache misses despite having no evictions. CanYing's lifelong opponent, CannotYing, wants to solve this problem so no one will choose CanYing's design anymore. Provide a detailed solution.

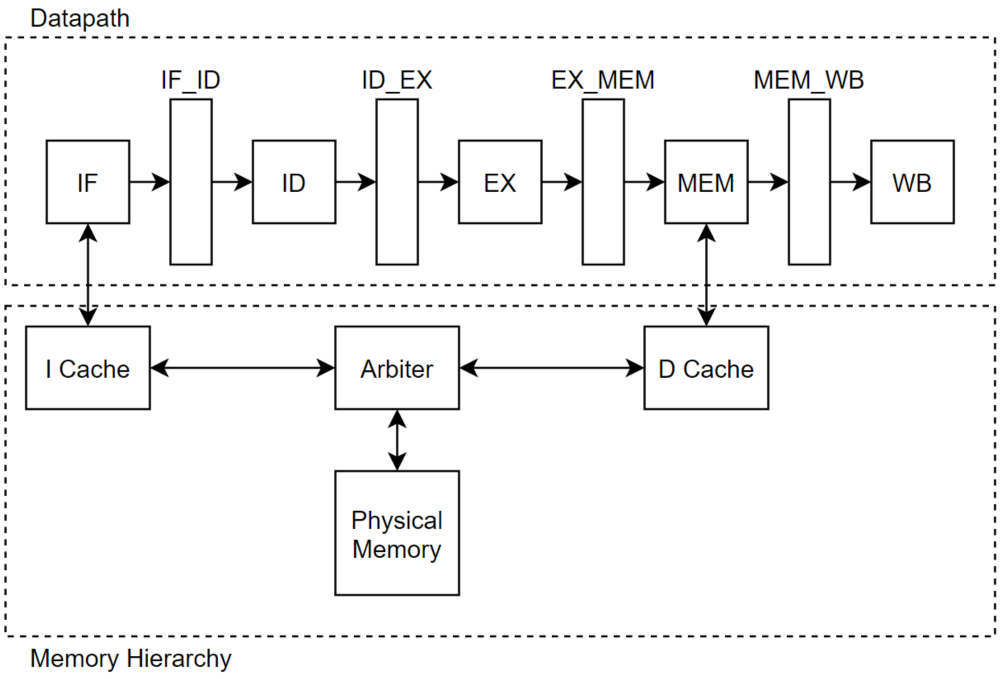
Prefetching. At the beginning a stride prefetcher or Markov prefetcher for instruction cache.

2. CannotYing argues that several optimizations are made in her new in-order pipeline design to save idle cycles due to serialized accesses to memory. Her motivation is that stalling the entire pipeline for any cache miss is expensive, thus she changed it to partial stalling where the "cache stages" will only stall any stage before them and insert NOP to the stage after them.

a. CanYing waited so long for revenge, willing to pay you points for achieving his goal. Provide a reason or a case to CanYing why CannotYing's design cannot "Ying".

IF & MEM only stall their own stages & insert NOP to the next stage. When both cache have a miss, NOP inserted by IF will overwrite instructions.

b. After CanYing went into bankruptcy because he paid too much for your help, you turned to CannotYing. CannotYing needs to fix the design before it comes to production to save her credibility. Propose a fix to this design on the below datapath. Note that there may be multiple answers for part a, you only need to fix the case you answered. You don't need to worry about whether your solution will fix other problems. You can elaborate on your solution below the datapath.



Stalling every stage before it.

3. CannotYing chooses to implement an out-of-order processor hoping to receive performance benefits. After running tests, she discovers that her application has low instruction-level parallelism and mostly comprises memory accesses to the same array.

a. CannotYing investigates further and finds that her in-order load-store queue is causing her processor to stall aggressively. What is another method that CannotYing can use to speed up memory accesses?

Since the application is mainly accessing the same array with the same addresses, she can implement a load-store buffer at the end of the load-store queue to store the most recent memory accesses and their values. (Changing the implementation of the load-store queue would not work due to low ILP).

b. After successfully cutting down memory latency, CannotYing finds that she is failing timing constraints due to a combinational path between her instruction queue, register-file, and reservation stations when issuing an instruction. CannotYing currently has implemented her processor with 1 instruction queue, 1 register-file, and every functional unit has its own reservation station. The processor has 3 adders, 2 multipliers, 2 dividers, and 1 load-store queue. How can CannotYing cut down her longest combinational path?

    Instead of having a reservation station for every functional unit, she uses the same reservation station for each type of functional unit. One reservation station for all of the adders etc.

c. CannotYing finds that despite the many for loops in the application, additional inconsistent branches are causing her pattern-history branch predictor to fail and flush the pipeline. How can CannotYing change her branch predictor to be more accurate for her processor?

She can implement a tournament branch predictor between a g-share branch predictor and her pattern-history predictor.

d. CannotYing notices that the main issue with her out-of-order pipeline is that certain instructions stall the pipeline while newer instructions are ready to commit. This can lead to the ROB getting filled up. To address this problem, CannotYing creates a system of committing all of the instructions that are ready to commit as a group. How would you handle interrupts in the given system?

Potpourri

Jingles are a great way to be remembered. Some of the greatest examples:

McDonald's "Ba-da-ba-ba-baaa ... I'm Lovin' It"

Kit Kat® "Give Me a Break... break me off a piece of that Kit Kat bar"

State Farm "Like a Good Neighbor, State Farm is there"

Huggies "Mommy, wow! I'm a Big Kid Now"

Craft a catchy jingle for ECE411.