1. Consider the following computation:

```
int x[256], y[256], w[256][256], s[3];

while (true) {
  for (i=0;i<256;i++) { // loop A
     x[i]=input();
     y[i]=0;
    }
  for (i=0;i<256;i++)   // loop B
   for (j=0;j<256;j++)
     y[j]+=x[i]*w[i][j];
  for (i=0;i<256;i++)   // loop C
     s[2]=max(y[i],s[2]);
     s[1]=max(s[1],s[2]);
     s[0]=max(s[0],s[1]);
  for (i=0;i<2;i++)    // loop D
     output(s[i]);
}
```

Original intent was to output all 3; typo in given problem with loop bound of 2 instead of 3.

- The initial input() provides a new input every 100 ns
- multiply is 5 ns operation, pipelineable to start one multiply every 1 ns
- local memory access (load, store) to w[][], x[], y[], s[] is 1 ns
- add and `max` are 1 ns operations
- ignore loop and indexing costs for this problem

(a) How many operations (load, store, add, max, multiply) in each labelled (A, B, C, D) for loop?

Loop	A	B	C	D
Operations	512 or 256	5×2^{16}	$4 \times 3 \times 2^8$	2 or 3

(b) Where is the bottleneck in this computation?

Loop B

(c) What is the Amdahl's law speedup if only the bottleneck is accelerated?

Assuming 100 ns is just time before guaranteed there is a next input: $\frac{5\times256+2+12}{2+12} \approx 92$

If you assume that input operation takes 100 ns: $\frac{5\times256+101+12}{101+12} \approx 12$

(d) What parallelism can be exploited in this task (both within and among loops)? Describe all applicable options where appropriate.

Loop	Parallelism Options
among loops	coarse-grained pipeline parallelism (across all)
A	(nothing really, bottlenecked on input)
B	data parallel (both task and SIMD) pipeline, VLIW
C	pipeline VLIW with software pipelining
D	could SIMD read but not really have any need

(e) Describe how you would speedup this task so that it can consume one input every 100 ns, limited only by the input rate.

- Input gives us a goal of handing 1 input per 100 ns
- Start by running each loop as a separate, coarse-grain dataflow pipeline task.
- This gives a goal of performing each (outer) loop iteration in less than 100 ns.
- Only loop B needs acceleration
  - A: Clearing y[i] can occur while waiting for next input to show up
  - C: 12 operations take 12ns, so can be pefrormed sequentially within the 100 ns cycle of the input
  - D: The output here is also not a rate limiter (needs even less than one output per 100 ns input)
- Loop B
  - Need to perform 256×5 operations every 100 ns to match the rate of the input. $\frac{256 \times 5}{100} = 12.8$ operations per nanosecond (timing on each operator).
  - So, we need to perform at least 13 operations per 1 ns cycle. Probably best to round that to 16.
  - Operations are SIMD and a multiple of 16, so a 16-lane vector unit used to accelerate the inner loop would do the trick.
  - Alternately, could break this into 16 parallel tasks, each computing a different set of 16 y[i]'s.

2. Consider the following computation:

```
int Image[1024][1024], Model[3][1024][1024], wpixel[1024][1024];
boolean mpixel[1024][1024];
for (y=0;y<1024;y++)
  for (x=0;x<1024;x++) {
     int pixel=Image[y][x];
     int M0=Model[0][y][x];
     int M1=Model[1][y][x];
     int M2=Model[2][y][x];
     mpixel[y][x]=f(pixel,M0,M1,M2); // 10 mpy, 6 adds
     int mupdate=g(pixel,M0,M1,M2); // 4 mpy, 10 adds
     int updateval=h(pixel,M0,M1,M2); // 16 mpy, 8 adds
     Model[mupdate][y][x]=updateval;
     }
for (i=0;x<1024;i++) { // ERRORs: x<1024 should be i; assignments should be wpixel
  if (mpixel[0][i]) mpixel[0][i]=1 else mpixel[0][i]=0;
  if (mpixel[i][0]) mpixel[i][0]=1 else mpixel[i][0]=0;
  }
for (y=1;y<1024;y++)
  for (x=1;x<1024;x++) {
     int imax=max(wpixel[y-1][x-1],max(wpixel[y-1][x],wpixel[y][x-1]));
     if (mpixel[y][x]) wpixel[y][x]=imax+1; else wpixel[y][x]=0;
     }
int xmax=0;
int ymax=0;
int maxval=0;
for (y=1;y<1024;y++)
  for (x=1;x<1024;x++)
     if (wpixel[y][x]>maxval) {maxval=wpixel[y][x]; xmax=x; ymax=y;}
int sy=max(0,ymax-16);
int sx=max(0,xmax-16);
for (y=sy;y<sy+16;y++)
  for (x=sx;x<sx+16;x++)
    output(Image[y][x]);
```

- Main memory is 256 M 32b ints; has a read and write latency of 100 ns, but can stream sequential data at 1 ns per cycle for blocks up to 512 words.
  `streamIn(MainAddr,LocalAddr,n)` – copy n≤512 32b ints to local memory in 100+n ns.
  `streamOut(LocalAddr,MainAddr,n)` – copy n≤512 32b ints to main memory in 100+n ns.
- Local memory is 4K 32b ints and has a read/write latency of 1 ns.
- multiply, add, max, compare each take 1 ns.
- As written Pixel, Model, mpixel, and wpixel live in main memory.
- Ignore loop and indexing costs for this problem.

(a) With no memory streaming operations or local memories,

i. estimate runtime
$2^{20}\,(6 \times 100 + 54 \times 1) + 2^{10}\,(2 \times 2 \times 100) + 2^{20}\,(5 \times 100 + 3 \times 1) + 2^{20}\,(2 \times 100 + 1) + 4 + 2^8 \times 100 \approx 1.4\text{B}$

ii. identify bottleneck (which loop? memory or compute?) and support your answer using your runtime estimate
First loop, memory

(b) Rewrite the code to localize and stream data You may combine loops where you find it beneficial.

i. Identify the local variables you define and how they are laid out in the local memory:

Address		
begin	end	Variable
0	2047	localM[0]
2048	4095	localM[1]
4096	6143	localM[2]
6144	10239	local_wpixel_row
10240	12287	localImage
12288	12291	prevx
12292		prevy
12296		prevxy
12300		curr_mpixel
12304		curr_wpixel
12308		lmpixel
12312		mupdate
12316		updateval
12320		pixel
12324		M0
12328		M1
12332		M2
12336		imax
12340		xmax
12344		ymax
12348		maxval
12352		sx
12356		sy
12360		x
12364		y

ii. Show how the code is revised to use these local variables and stream fetch operations.

```
int Image[1024][1024], Model[3][1024][1024];
int localM[3][512]; // was erroneously 1024 on original solution
int local_wpixel_row[1024];
int xmax=0;
int ymax=0;
int maxval=0;
int curr_mpixel, curr_wpixel;
int prevx, prevy, prevxy;
for (int y=0;y<1024;y++)
  for (int xb=0;xb<1024;xb+=512) {
    streamIn(&Image[y][xb],localImage,512);
    streamIn(&Model[0][y][xb],&localM[0],512);
    streamIn(&Model[1][y][xb],&localM[1],512);
    streamIn(&Model[2][y][xb],&localM[2],512);
    for (int xoff=0;xoff<512;xoff++) {
       int x=xb+xoff;
       int pixel=localImage[xoff];
       int M0=localM[0][xoff];
       int M1=localM[1][xoff];
       int M2=localM[2][xoff];
       curr_mpixel=f(pixel,M0,M1,M2); // 10 mpy, 6 adds
       int mupdate=g(pixel,M0,M1,M2); // 4 mpy, 10 adds
       int updateval=h(pixel,M0,M1,M2); // 16 mpy, 8 adds
       localM[mupdate][xoff]=updateval;
       if ((y==0) || (x==0)) {
             prevy=0;
             curr_wpixel=0;
             imax=0;
          }
       else {
          prevy=local_wpixel_row[x];
          int imax=max(prevxy,max(prevy,prevx));
          }
       if (curr_mpixel) curr_wpixel=imax+1; else curr_wpixel=0;
       local_wpixel_row[x]=curr_wpixel;
       prevx=curr_wpixel;
       prevxy=prevy;
       if (curr_wpixel>maxval) {maxval=curr_wpixel; xmax=x; ymax=y;}
       }
    streamOut(&localM[0],&Model[0][y][xb],512);
    streamOut(&localM[1],&Model[1][y][xb],512);
    streamOut(&localM[2],&Model[2][y][xb],512);
```

```
     }
int sy=max(0,ymax-16);
int sx=max(0,xmax-16);
for (int y=sy;y<sy+16;y++)
  {
     streamIn(&Image[y][x],&localImage,16);
     for (int x=sx;x<sx+16;x++)
         output(localImage[x]);
   }
```

10 points for streaming in/out first loop
10 points for optimizing rest; full points for localizing mpixel, wpixel row; 5 points if keep image size mpixel/wpixel and use streaming on those.

(c) What is the runtime of your optimized design?

$2^{20}(69) + 2^{11}(612 \times 7) + 4 + 16 \times 116 + 2^8 \times 1 \approx 81\text{M}$

(d) Where is the bottleneck now?

Computation in first loop

3. Consider a function from $A00$, $A01$, $A10$, $A11$, $B0$, $B1$ to $B2$, $B3$:

$$t0 = \frac{A00}{A10} \quad (1)$$
$$t1 = \frac{A01}{A11} \quad (2)$$
$$t2 = t1 * B1 \quad (3)$$
$$t3 = B0 - t2 \quad (4)$$
$$t4 = t1 * A10 \quad (5)$$
$$t5 = A00 - t4 \quad (6)$$
$$t6 = t0 * B1 \quad (7)$$
$$t7 = B0 - t6 \quad (8)$$
$$t8 = t0 * A11 \quad (9)$$
$$t9 = A01 - t8 \quad (10)$$
$$t10 = \frac{t3}{t5} \quad (11)$$
$$t11 = \frac{t7}{t9} \quad (12)$$
$$t12 = A20 * t10 \quad (13)$$
$$t13 = A21 * t11 \quad (14)$$
$$t14 = A30 * t10 \quad (15)$$
$$t15 = A31 * t11 \quad (16)$$
$$B2 = t12 + t13 \quad (17)$$
$$B3 = t14 + t15 \quad (18)$$

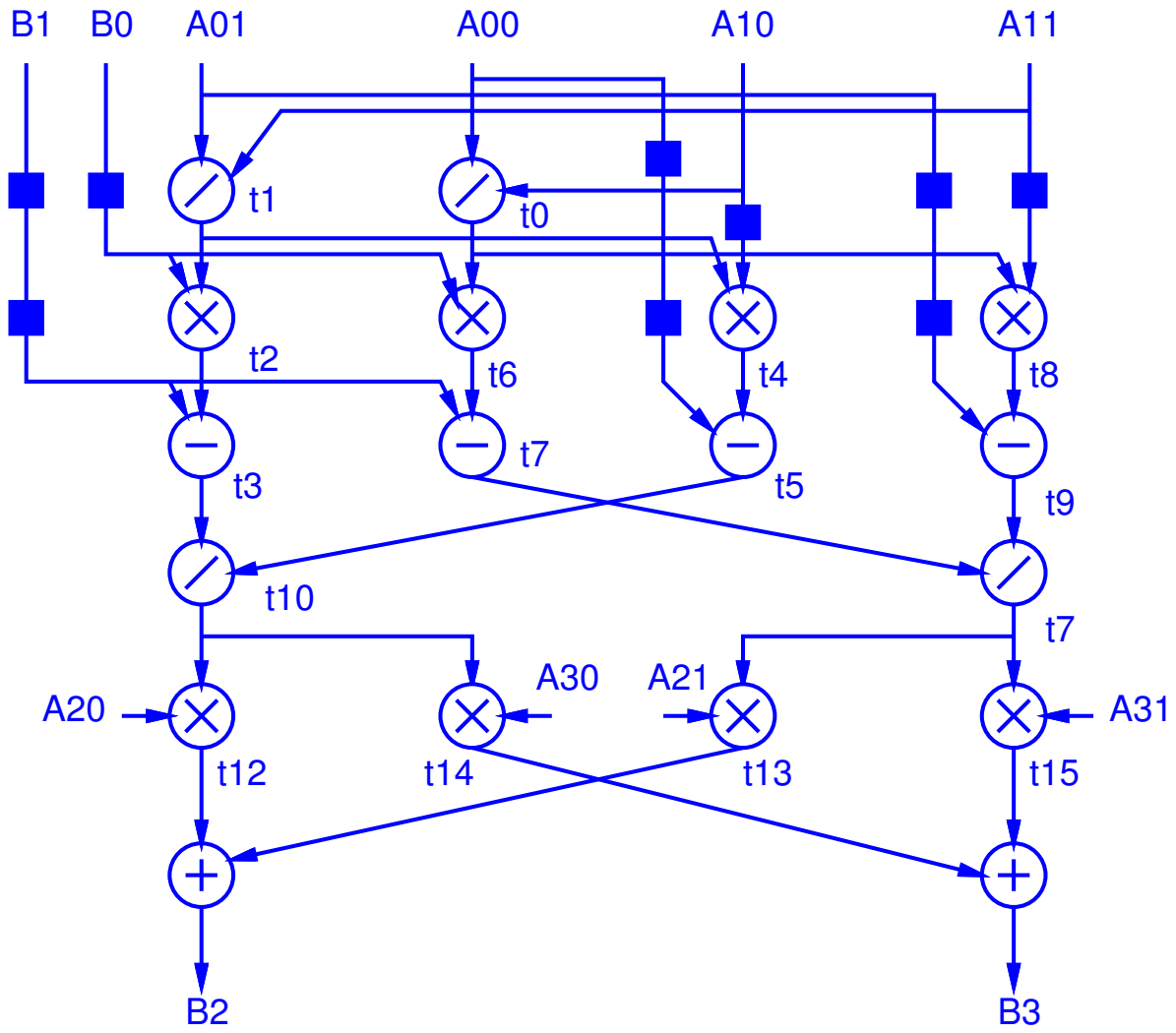
Assume:

- $A00$, $A01$, $A10$, $A11$, $B0$, $B1$ available on inputs at beginning of cycle
- output $B2$, $B3$ on designated output port
- $A20$, $A21$, $A30$, $A31$ already in operator memories; you choose which
- add/subtract, multiply, divide are single-cycle operations
- add/subtract unit costs 1 units of area
- multiply unit costs 10 units of area
- divide unit costs 10 units of area
- memory bank costs 5 units of area
- $i \times o$ crossbar costs $0.5 \cdot i \cdot o$ units of area
- word-wide pipeline register costs 0.5 units of area
- 2 or 3 input mux is 1 unit of area

(a) What is the critical path bound for this computation?

6 cycles

(b) Show a pipelined datapath for this operation.

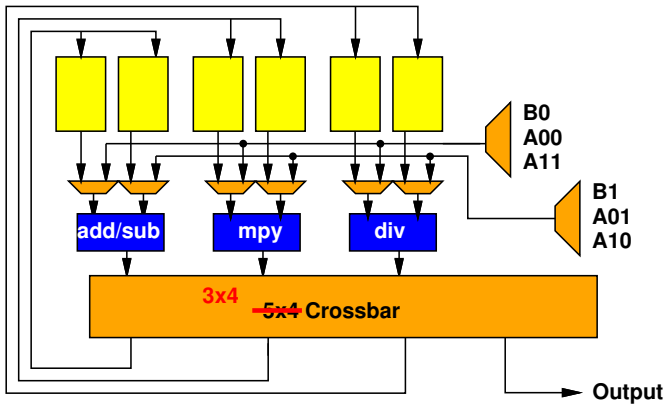


Note registers added (small squares) to balance path delays from inputs. As shown, assumes a register on the output of each operator.

(c) Estimate the area for the pipelined datapath.

$(8+4)\,10 + 1 \times 6 + \frac{9}{2} = 130.5$

Problem statement wasn't clear if there were registers associated with each operator, so may add 16 more registers for the ones associated with the operators. Maybe also add registers for A20, A21, A30, A31. So, +16 or +20 register (+8 or +10 area units) also reasonable.



(d) What is the resource bound for this computation on a VLIW datapath with a single add/subtract unit, a single multiplier, and a single divider (as shown)?

$\max\left(\frac{6}{1}, \frac{8}{1}, \frac{4}{1}\right) = 8$

(e) Schedule the computation on the VLIW datapath with a single add/subtract unit, a single multiplier, and a single divider (as shown) to minimize computation cycles.
Mark each "operator" with the variable computed on the operator on that cycle; mark each "input" with the variable being stored into the data memories on each cycle (note: only one value can be stored into the data memories associated with an operator on each cycle).

Cycle	**add/sub**		**multiply**		**divide**		**mux0**	**mux1**	**output**
	operator	input	operator	input	operator	input			
0				t1	t1		A11	A01	
1				t0	t0		A00	A10	
2		t2	t2				B0		
3		t4	t4					A10	
4	t3	t6	t6			t3	B0		
5	t5	t8	t8			t5	A00	A11	
6	t7			t10	t10	t7	B0		
7	t9	t12	t12			t9		A01	
8		t14	t14	t11	t11				
9		t13	t13						
10	B2	t15	t15						B2
11	B3								B3
12									
13									
14									
15									

(f) Estimate the area of the VLIW datapath with a single add/subtract unit, a single multiplier, and a single divider (as shown).

$5 \times 6 + 10 + 10 + 1 + 8 + \frac{3 \times 4}{2} = 65$

(g) Using no more than 100 units of area, provision a customized VLIW datapath for this unit – how many operators of each type? total area?

Operator	add/sub	mpy	div	mux2 or mux3	Area
Number	1	2	1	2 (10)	91

2 muxes as input select; 10 total including the pair for each of the 4 operators.

$5 \times 8 + 10 + 2 \times 10 + 1 + 10 + \frac{5 \times 4}{2} = 91$

(h) Justify your choice of operators.

Resource Bound $\max\left(\frac{6}{1}, \frac{8}{2}, \frac{4}{1}\right) = 6$. Multiply is the bottleneck resource, so priority to add. There is only enough space to add one operator.

A schedule would even be better, but was not expected.

Arguments about maximum parallelism available (maximum number of operators that could be usefully employed) were good, but those only tended to be applicable for designs that do not meet the area constraint (and are larger than the fully pipelined design).

1. Consider the following set of real-time tasks for a vision-based control system:

Task	Frequency	Operations per Invocation
Plan	once per second	1.28×10^{11}
Control	100 times per second	10^6
Mapping	50 times per second	4×10^7
Collide	100 times per second	10^7

Consider the following computational elements:

Compute Element	Performance	Area
High Performance Processor (HPP)	1 op per cycle at 2 GHz	$3{\times}10^4$
VLIW	8 ops per cycles at 500 MHz	$3{\times}10^3$
6-LUT (with interconnect)	200 MHz	1

Consider the following spatial datapaths for each of the tasks:

Task	6-LUTs	Effective operations per cycle
Plan	1,300	40
Control	32,000	500
Mapping	33,000	1024
Collide	3,200	100

Assume:

- perfectly divide work among processors (if need more than one),
- perfectly schedule VLIW
- perfectly divide task among instances of spatial datapaths shown above
- may share processor or VLIW among multiple tasks

(a) Complete table with the number of homogeneous resources required to meet real-time task goals and the associated area: (May share processors, VLIW, but must pay for an integer number of each.)

Task	HPP	VLIW	6-LUTs
Plan	$\frac{1.28\times10^{11}}{2\times10^{9}}$=64	$\frac{1.28\times10^{11}}{8\times5\times10^{8}}$=32	$\left(\frac{1.28\times10^{11}}{40\times2\times10^{8}}\right)$ 1300=20,800
Control	$\frac{100\times10^{6}}{2\times10^{9}}=\frac{1}{20}$	$\frac{100\times10^{6}}{8\times5\times10^{8}}=\frac{1}{40}$	$\left\lceil\frac{100\times10^{6}}{500\times2\times10^{8}}\right\rceil$ 32000=32,000
Mapping	$\frac{50\times4\times10^{7}}{2\times10^{9}}=1$	$\frac{50\times4\times10^{7}}{8\times5\times10^{8}}=\frac{1}{2}$	$\left\lceil\frac{50\times4\times10^{7}}{1024\times2\times10^{8}}\right\rceil$ 33000=33,000
Collide	$\frac{100\times10^{7}}{2\times10^{9}}=\frac{1}{2}$	$\frac{100\times10^{7}}{8\times5\times10^{8}}=\frac{1}{4}$	$\left\lceil\frac{100\times10^{7}}{100\times2\times10^{8}}\right\rceil$ 3200=3,200
Total (integer)	66	33	89,000
Total Area	2.0×10^{6}	9.9×10^{4}	8.9×10^{4}

(b) Heterogeneous solution that meets real-time goals and minimizes area. For each task identify the number and type of compute units.
(May share processors, VLIW, but must pay for an integer number of each. Show number of datapaths for 6-LUT case, and calculate total area of those datapaths for Area column, when appropriate.)

Task	Type	Number	Area
Plan	6-LUT	16×1300	20,800
Control	VLIW	1	3,000
Mapping	VLIW		
Collide	VLIW		
		Total Area	23,800

(c) Show schedule for any HPP or VLIW processors that run multiple tasks at the coarsest time-slice for the task set.

Time Slot				Time Slot			
10 ms				10 ms			
5 ms	2.5 ms	0.25 ms	2.25 ms	5 ms	2.5 ms	0.25 ms	2.25 ms
map	collide	control	(unused)	map	collide	control	(unused)

Since these are real-time tasks, they must run at the specified frequency. We must run control and collide once every 10 ms (100 times per second). This means we must schedule all the tasks on the same processor on a 10 ms time slice to guarantee each control and collide task gets run. Since the map task only needs to run once ever 20 ms, we must split its $\frac{4\times10^7}{8\times5\times10^8\text{Hz}} = 10\,\text{ms}$ runtime into two 5 ms slices so that it gets its 10 ms of runtime every 20 ms while allowing control and collide to run once ever 10 ms.

2. Parallelism in Plan Task

   Working with the following algorithm for Plan and assuming this takes the 1.28×10^{11} operations stated in Problem 1:

```
#define DIM 1024
typedef struct cell {
   uint32_t cost;
   uint32_t path;
   uint16_t edge_cost[4];
   } cell_struct;
cell_struct from[DIM][DIM];
cell_struct to[DIM][DIM];
int xoff, yoff;

for (int iter=0;iter<3*DIM;iter++) {
  for (int y=0;y<DIM;y++)
    for (int x=0;x<DIM;x++) {
       uint32_t min_cost=from[y][x].cost;
       uint32_t min_path=from[y][x].path;
       for (int j=0;j<4;j++) {
         if (j==0)      {xoff=0; yoff=-1;}
         else if (j==1) {xoff=-1; yoff=0;}
         else if (j==2) {xoff=1; yoff=0;}
         else if (j==3) {xoff=0; yoff=1;}
         uint32_t ncost=from[y+yoff][x+xoff].cost;
         uint32_t ecost=from[y][x].edge_cost[j];
         uint32_t icost=ncost+ecost
         if (icost < min_cost){
            min_path=ENCODE(yoff,xoff);  // count 3 ops (depth 1 op)
            min_cost=icost;
            } // icost < min_cost
         } // for j
       to[y][x].cost=min_cost;
       to[y][x].path=min_path;
      } // for x
  cell_struct **tmp=from;
  from=to;
  to=tmp;
  } // for iter
```

(a) Describe how you can divide the computational task among processors or hardware datapaths to achieve various levels of parallelism Write code snippets or edits as necessary to make your division clear.

There are no data dependencies within the x and y loops, so computations are data parallel for each output [y][x] location. We can divide the computation by x-y regions onto the processors.

The simplest solution, which is adequate for the parallelism needed for Problem 1 (16–64), is to divide along y values, assigning each processor or datapath k to a range of y values from $\left(\frac{DIM}{P}\right)k$ to $\left(\frac{DIM}{P}\right)(k+1)-1$.

Dividing over both x and y would reduce the number of `from[y][x]` cells that are needed by multiple processors.

(b) Building on your answer above, how do you provide the specific parallelism needed in Problem 1? (Number is the value you calculated in Problem 1)

Compute Element	Number	How
HPP	64	Assign y=$16k$ to $16(k+1)-1$ to each processor k
VLIW	32	Assign y=$32k$ to $32(k+1)-1$ to each processor k
6-LUT datapaths	16	Assign y=$64k$ to $64(k+1)-1$ to each datapath k

(c) What is the Latency Bound (critical path) in operations for the entire task assuming the data needed lives in registers (i.e., no latency coming or going from memory)?

No x, y dependence, so all the j loops can be run concurrently. There is a dependence between outer (iter) iterations, so we must run the 3×DIM j-loops in series. Each j-iteration performs an addition (icost), comparison, and a selection (2 serial operations). The ENCODE can happen in parallel with the comparison (or, strictly speaking, can be performed once at the beginning of the computation outside the iter loop). The 4 icost additions have no dependence and can be performed in parallel. Serialized by the comparison/selection, the four iterations of the j-loop have a latency of $4 \times 2 + 1$ (also see solution to Problem 3a). So, the entire latency is: $3 \times 1024 \times (4 \times 2 + 1) = 27,648$.

The only reason the j-loop is serialized is that we are trying to guarantee the path selected is exactly the same as the sequential case when there are equal cost options from multiple directions. If we don't care which path is seleted, just that it is a minimum cost path, then we can perform the j-loop selection as a reduction rather than as a sequential operation. Now we can compute the ncost additions all at once, then 3 stages of reduce ($\lceil \log_2(5) \rceil$, since we must select among 5 things – the original and the 4 neighbors), each of which requires a comparison and a mux selection. So, the latency is: $3 \times 1024 \times (1 + 3 \times 2) = 21,504$.

Either solution was acceptable. In practice, if there's no reason to prefer a particular path, the lower latency (more parallel) one is preferred. If you are working with customers, you might ask their constraints. For an exam or assignment, it is always best to state the choice you made. Similarly, for customers, it is good to document decisions like this.

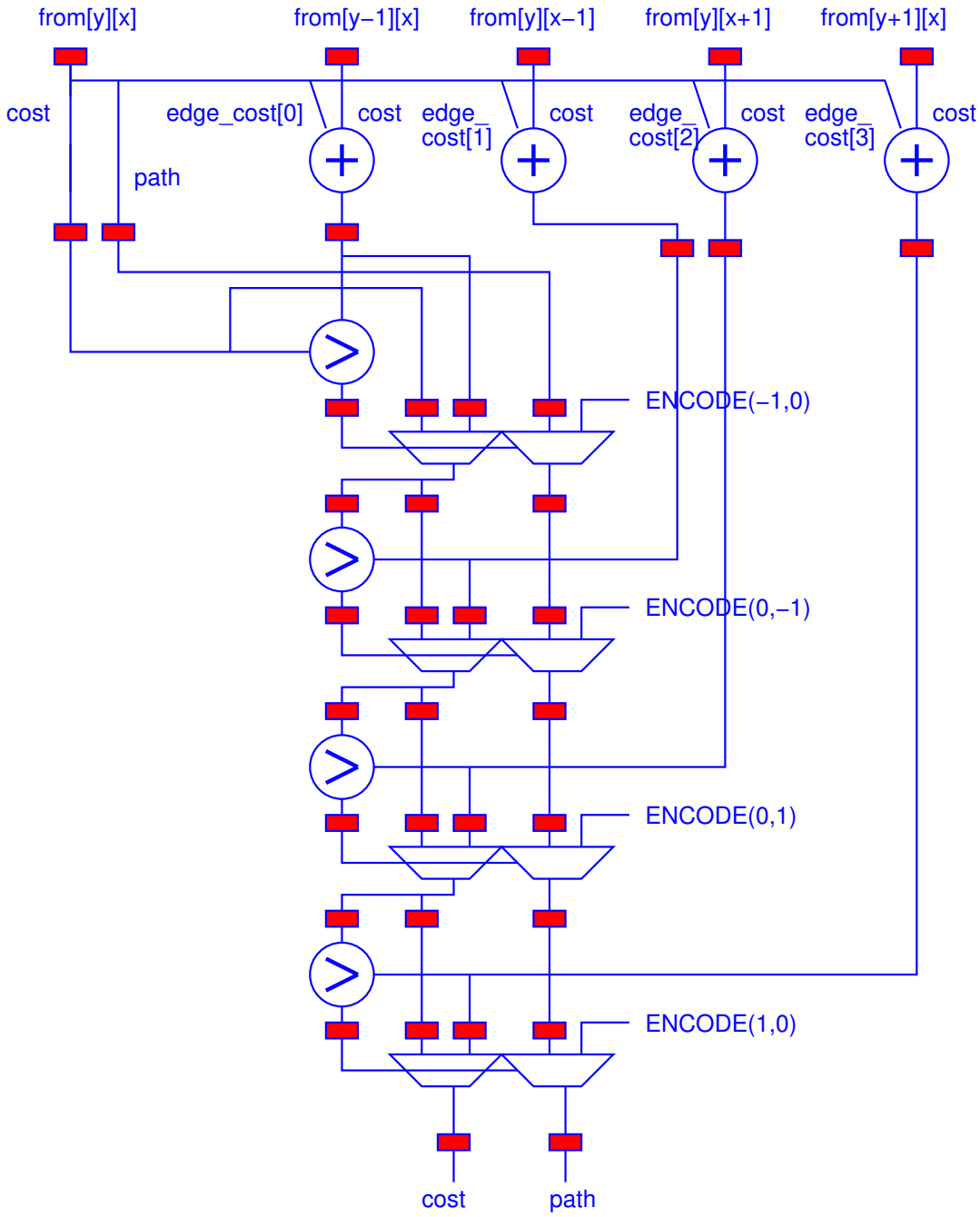
3. Hardware datapath for Plan

   Working with the same algorithm for Plan:

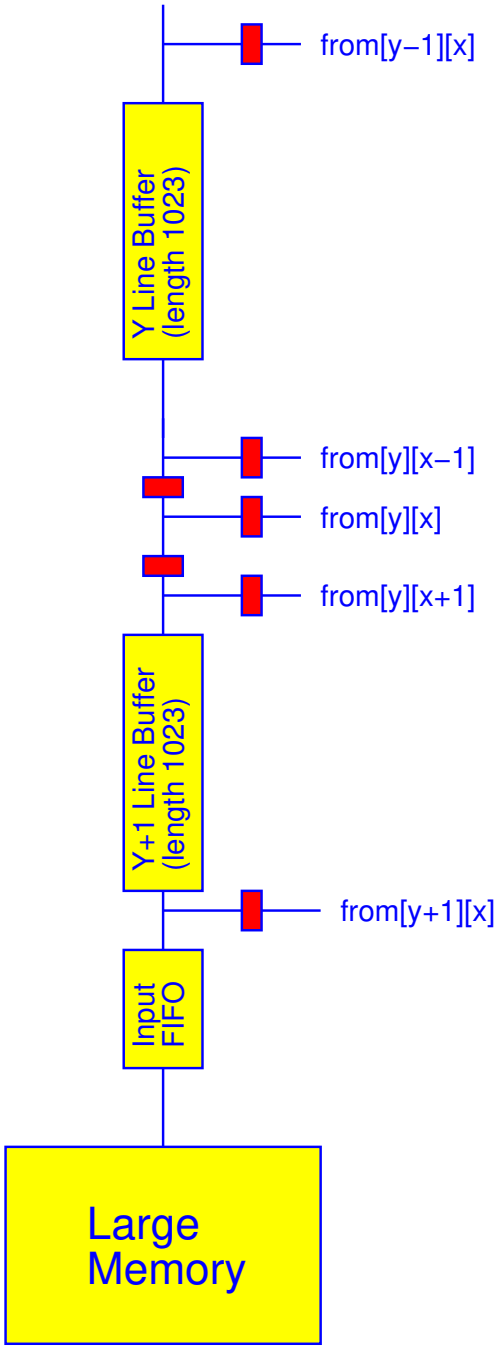
```
#define DIM 1024
typedef struct cell {
   uint32_t cost;
   uint32_t path;
   uint16_t edge_cost[4];
   } cell_struct;
cell_struct from[DIM][DIM];
cell_struct to[DIM][DIM];
int xoff, yoff;

for (int iter=0;iter<3*DIM;iter++) {
  for (int y=0;y<DIM;y++)
    for (int x=0;x<DIM;x++) {
       uint32_t min_cost=from[y][x].cost;
       uint32_t min_path=from[y][x].path;
       for (int j=0;j<4;j++) {
         if (j==0)      {xoff=0; yoff=-1;}
         else if (j==1) {xoff=-1; yoff=0;}
         else if (j==2) {xoff=1; yoff=0;}
         else if (j==3) {xoff=0; yoff=1;}
         uint32_t ncost=from[y+yoff][x+xoff].cost;
         uint32_t ecost=from[y][x].edge_cost[j];
         uint32_t icost=ncost+ecost
         if (icost < min_cost){
            min_path=ENCODE(yoff,xoff);  // count 3 ops (depth 1 op)
            min_cost=icost;
            } // icost < min_cost
         } // for j
       to[y][x].cost=min_cost;
       to[y][x].path=min_path;
      } // for x
  cell_struct **tmp=from;
  from=to;
  to=tmp;
  } // for iter
```

(a) Unroll and pipeline the inner (j) loop.
Draw the resulting pipelined datapath.



(b) Show how you manage data storage and movement of `from` data in order to deliver `from` data to the datapath above. Assume data is streamed from the large memory. Draw registers and memories and show inputs to the Part a datapath.



4. Data Managment

   Continuing to work with the algorithm for Plan:

```
#define DIM 1024
typedef struct cell {
   uint32_t cost;
   uint32_t path;
   uint16_t edge_cost[4];
   } cell_struct;
cell_struct from[DIM][DIM];
cell_struct to[DIM][DIM];
int xoff, yoff;

for (int iter=0;iter<3*DIM;iter++) {
  for (int y=0;y<DIM;y++)
    for (int x=0;x<DIM;x++) {
       uint32_t min_cost=from[y][x].cost;
       uint32_t min_path=from[y][x].path;
       for (int j=0;j<4;j++) {
         if (j==0)      {xoff=0; yoff=-1;}
         else if (j==1) {xoff=-1; yoff=0;}
         else if (j==2) {xoff=1; yoff=0;}
         else if (j==3) {xoff=0; yoff=1;}
         uint32_t ncost=from[y+yoff][x+xoff].cost;
         uint32_t ecost=from[y][x].edge_cost[j];
         uint32_t icost=ncost+ecost
         if (icost < min_cost){
            min_path=ENCODE(yoff,xoff);  // count 3 ops (depth 1 op)
            min_cost=icost;
            } // icost < min_cost
         } // for j
       to[y][x].cost=min_cost;
       to[y][x].path=min_path;
      } // for x
  cell_struct **tmp=from;
  from=to;
  to=tmp;
  } // for iter
```

- Processor has a local memory that holds 64KB with single cycle access for 16b, 32b, or 64b data.
- uint32_t is stored in 4 Bytes; uint16_t is stored in 2 Bytes.
- `from` and `to` live in the large memory.
- Large memory has a read latency of 50 ns (100 cycles on HPP) for a single 16b, 32b, or 64b word or 100 ns (200 cycles) for a 2048b transfer into the local memory
  - For the processor, assume you have a routine
    `void read2048(int *local_buf,int *large_mem_buf)`
    that will initiate a 2048b read from the large memory into the processor local memory at the specified addresses.
- The processor can issue a write to the large memory in a single cycle and continue without waiting.
  - We should be concerned with when a read may need something previously written to memory and the impact of writes on memory bandwidth. To keep this problem simple, you may ignore these effects.
- Assume scalar (non-array) variables can live in registers.
- You may ignore loop and conditional overheads in processor runtime estimates; additions and comparisons are single cycle operations.

(a) What time (in ns) is required for the task as written when running on the HPP and all references to each component of `from` go to the large memory?

Read from **`from`** 10 times inside x loop (cost, path, and 4 reads to each of cost and edge_cost inside the j loop):
$3 \times 1024^3 \times 10 \times 50$ ns.
Perform 3 adds, 1 compare, and 1 ENCODE (3 ops) inside j-loop contributing 7×4 operations inside x loop; perform 2 writes to **`to`** for a total of 30 cycles inside x-loop:
$3 \times 1024^3 \times 30 \times 0.5$ ns.
Total: $3 \times 1024^3 \times 515$ ns$=1.7 \times 10^{12}$ ns.

(b) What fraction of time is spent performing read operations from the large memory for `from`?

$\frac{500}{515} = 97\%$

(c) What is the Amdahl's Law speedup if you only accelerate these memory reads to `from`?

$\frac{515}{15}=34\times$

(d) How do you need to modify the design to minimize the time spent reading `from` from the large memory?
Describe how the code needs to be modified. Show revised code as necessary for clarity.

Keep the the y-1, y, and y+1 lines in local memory. With 16B per cell_struct, this is 1024×16B per line or 48KB for all 3 lines, which fits in the 64KB memory. On entering the y loop, we will need to read in a new line. We can do this efficiently using read2048 to read 16 cell_struct entries at a time (16 entries $\times$ 16B/entry=256B $\times$ 8b/B = 2048b). Inside the x loop, all references are to the local memory. To get things setup, we need a prologue to read the first two lines before the y loop.

```
// all 3 in local memory
cell_struct prev[DIM];
cell_struct current[DIM];
cell_struct next[DIM];

for (int iter=0;iter<3*DIM;iter++) {
  for (int x=0;x<DIM;x++)
     current[x]=OFF_CELL; // cell with maxval for .cost so never chosen
  for (int x=0;x<DIM;x+=(256/sizeof(cell_struct)))
     read2048(&next[x],&from[0][x]);
  for (int y=0;y<DIM;y++) {
    cell_struct *tmp=prev;
    prev=current;
    current=next;
    next=tmp;
    for (int x=0;x<DIM;x+=(256/sizeof(cell_struct)))
      read2048(&next[x],&from[y+1][x]);
    for (int x=0;x<DIM;x++) {
       uint32_t min_cost=current[x].cost;
       uint32_t min_path=current[x].path;
       for (int j=0;j<4;j++) {
         cell_struct *nline;
         if (j==0)      {xoff=0; yoff=-1; nline=prev;}
         else if (j==1) {xoff=-1; yoff=0; nline=current;}
         else if (j==2) {xoff=1;  yoff=0; nline=current;}
         else if (j==3) {xoff=0;  yoff=1; nline=next}
         uint32_t ncost=nline[x+xoff].cost;
         uint32_t ecost=current[x].edge_cost[j];
         uint32_t icost=ncost+ecost
         if (icost < min_cost){
            min_path=ENCODE(yoff,xoff);  // count 3 ops (depth 1 op)
            min_cost=icost;
            } // icost < min_cost
         } // for j
       to[y][x].cost=min_cost;
       to[y][x].path=min_path;
      } // for x
    } // for y
  cell_struct **tmp=from;
  from=to;
  to=tmp;
  } // for iter
```

(e) What is the time (in ns) required on a single HPP processor after this modification?

Each read2048 is able to bring in 256B. Since each cell_struct is 16B, this means we get 16 cell_structs per 100 ns read.

Reading `from`: $3 \times 1024^2 \times \frac{1024 \times 16}{256} \times 100\,\text{ns}$.

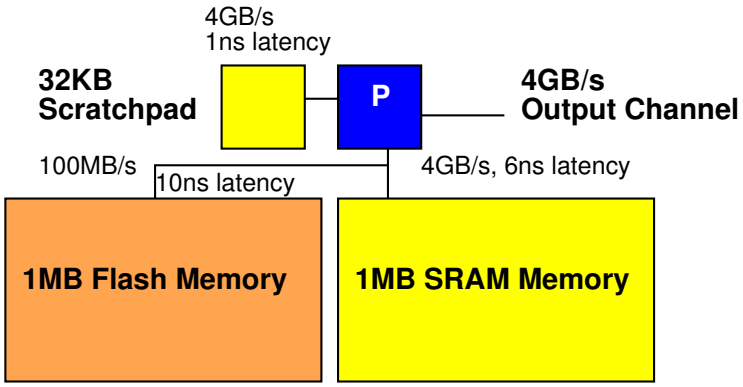
Local Reads: $3 \times 1024^3 \times 10 \times 0.5\,\text{ns}$.

Operations and writes: $3 \times 1024^3 \times 30 \times 0.5\,\text{ns}$.

Total: $3 \times 1024^3 \times 26.25\,\text{ns} = 8.5 \times 10^{10}\,\text{ns}$.

```
// You will be determining a value for FREQBYTES
#define WINDOW 1024
#define MAXBITLEN 11
#define LOG_MAXBITLEN 4
#define MAX_FREQS 255
#define MASKLOOKUP ((1<<MAXBITLEN)-1)
#define MASKLEN ((1<<LOG_MAXBITLEN)-1)
#define AMPLEN 14
#define FREQLEN 14
#define MASKAMP ((1<<AMPLEN)-1)
#define MASKFREQ ((1<<FREQLEN)-1)
uint8_t in[FREQBYTES];
uint32_t fa[FREQS];
uint32_t lookup[1<<MAXBITLEN];
uint16_t s[MAX_FREQS][WINDOW];
while(1) { // Outer while loop
    uint32_t ts[WINDOW];
    for (j=0;j<WINDOW;j++) ts[j]=0; // Loop A
    uint8_t freqs=read_flash_byte(); // max rate 100MB/s
    for(int i=0;i<FREQBYTES;i++) // Loop B
        in[i]=read_flash_byte();
    uint11_t top11=((int *)in)[0]>>21;
    uint11_t next11=(((int *)in)[0]>>10)&MASKLOOKUP;
    int next11bitpos=11;
    for(i=0;i<freqs;i++) { // freqs<MAX_FREQS // Loop C
      uint32_t res=lookup[top11];
      uint32_t tfa=res>>LOG_MAXBITLEN; fa[i]=tfa;
      uint4_t len=MASKLEN & res;
      uint32_t t1=(top11<<len); uint4_t t2=(MAXBITLEN-len); uint32_t t3=(next11>>t2);
      top11= t1|t3;
      next11bitpos+=len;
      uint32_t bytepos=next11bitpos>>3; uint3_t bitoffset=next11bitpos%8;
      uint32_t wordval=(*((int *)(&in[bytepos]))); // treat as 1 cycle
      uint4_t t4=(21-bitoffset); uint32_t t5=(wordval>>t4);
      next11=MASKLOOKUP & t5;
      }
   for (i=0;i<freqs;i++) { // Loop D
       uint16_t freq=(fa[i]>>AMPLEN) & MASKFREQ;
       uint16_t amp=fa[i] & MASKAMP;
       for (j=0;j<WINDOW;j++) // Loop E
           ts[j]+=s[freq][j]*amp;
       }
   for (j=0;j<WINDOW;j++) // Loop F
       output(ts[j]); // max rate 4GB/s
}
```

We start with a baseline, single processor system as shown.



- For simplicity throughout, we will treat non-memory indexing adds (subtracts count as adds), shifts, mod-by-power-of-two, ORs, ANDs, and multplies as the only compute operations. We'll assume the other operations take negligible time or can be run in parallel (ILP) with the adds, multiplies, and memory operations. (Some consequences: You may ignore loop and conditional overheads in processor runtime estimates; you may ignore computations in array indecies.)
- Assume all additions are associative.
- Baseline processor can execute one compute operation (above) per cycle and runs at 1 GHz.
- Constant expressions (like $1 << 8$) are evaluated by the compiler and take no time to compute at runtime.
- Maximum data rate for reading from flash is 100MB/s. Latency of read is 10 ns.
- The output port used by `output()` can transfer data at 4GB/s (one 32b word per cycle at 1 GHz).
- Baseline processor has a 32KB local scratchpad memory.
- in[], fa[], ts[], and lookup[] fit in the local scratchpad memory close to the processor and can be read or written in a single cycle.
- For the baseline processor, s[] lives in the large (1MB) memory and requires 6 cycles to access.
- lookup[] and s[] are prepopulated with content before entering the while loop (not shown).
- Assume adds and multiplies take 1 ns when implemented in hardware accelerator, so fully pipelined accelerators also run at 1 GHz.

1. For sequential evaluation and assuming FREQBYTES is 256.

   (a) Worst-case cycles to compute one iteration of the outer while loop?
   (show cycles per loop for partial credit consideration.)

A	WINDOW	1024
B	FREQBYTES×10 100 MB/s bandwidth=10 cycles/byte	2560
between	5	5
C	15 × MAX_FREQS 12 ops, 3 scratchpad memory accesses	3825
D, E	MAX_FREQS × (5 + WINDOW × 10) E 10: 6 for read from s[][] + read and write ts[] + multiply, add	2,612,475
F	WINDOW	1024
Total		2,620,913

   2.6 million cycles

   (b) Which outer loop is the bottleneck?

   Circle One:

A	B	C	(D)	F

   (c) What is the Amdhal's Law maximum speedup for accelerating the identified loop?

   $\frac{A+B+C+D+F}{A+B+C+F} = \frac{2{,}620{,}913}{2{,}620{,}913-2{,}612{,}475} = 310$

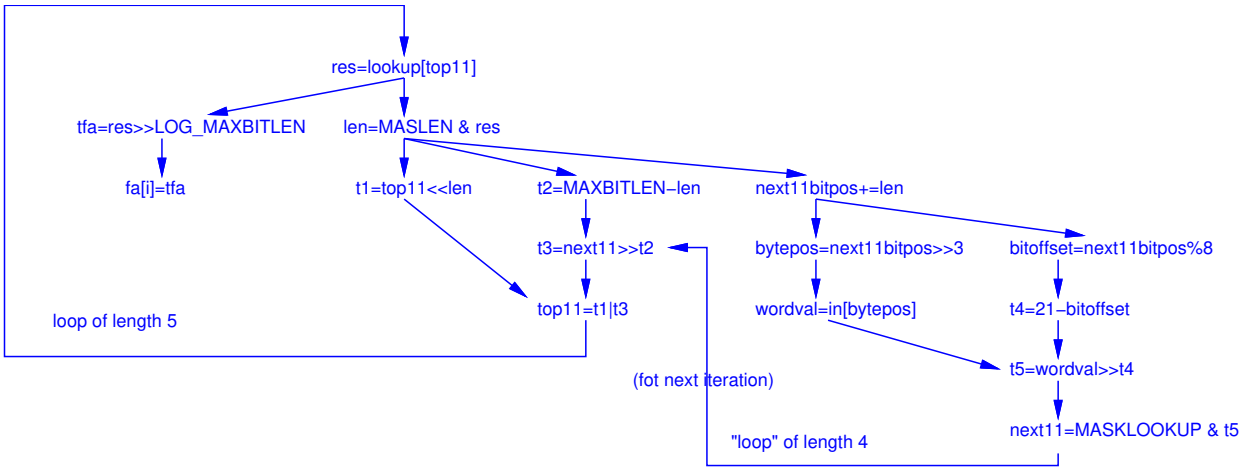
2. Loop C

   (a) How many memory operations does one instance of the loop perform?
   3 – lookup[], fa[], in[]

   (b) How many compute operations (of the set identified) does the loop perform?
   12

   (c) Assuming unlimited compute operators and memory ports, what is the minimum achievable Initiation Interval (II) for this loop?
   Draw dataflow graph and identify any data-dependent loops for full credit.

II=5



Note: Critical path is 7. The key loop is the one around top11, which is of length 5. We must also be able to update next 11, and that is in a loop of length 4. Strictly, it's not a loop itself, but we do need to be able to compute the next11 within one II, and this does fit.

3. Data Parallel: Classify Loops C, D, and E:

Loop	Data Parallel?	Associative Reduce?	Must be Sequential?
C			Yes
D		Yes	
E	Yes		

C: The dependent loop for top11 identified in Problem2c forces sequentialization of the loop.

E: operations are independent for each j. Can perform the entire multiplication and add concurrently. This vectorizable.

D: If you think about unrolling E into a vector, then unrolling D as well, the only dependency is the add chain for each `freq` into ts[j]. The add is associative, so this is an associative reduce operation.

4. What is the latency bound for executing Loops C and D (from the beginning of C to the end of D)?

   - assume memories of unbounded width (no bandwidth limits)
   - respect latencies for memory access

   Loop C: From Problems 2 and 3, we know this loop is sequentially dependent with an II of 5. So, it will take: MAX_FREQS $\times$ II=255$\times$5=1275 cycles.

   Loop E: This is data parallel. Fully unrolled this takes 6 (read s[][])+1=7 cycles to get to the products.

   Loop D: This is a reduce add across MAX_FREQS values to produce each ts[j]. That can be done in $\log_2(MAX\_FREQS) =$ 8 cycles. There's a final write into ts[j] at the end.

   Together, this gives us 1275+7+8+1=1291 cycles or 1.3μs.

   We can do slightly better observing that we can overlap some (or most) of the additions in D-E with C. So, even if we sequentially perform the E vector adds, we can complete one per cycle and match pace with C. So, after finishing C, we only need to perform the 7 cycles for the E lookup and multiply, then a final add and store So, we can perform this is 1275+7+1+1=1284. To two significant figures, this is also 1.3μs.

5. Data Streaming: How big (minimum size) does the buffer need to be between the identified loops in order to allow the loops to profitably execute concurrently.
(Hint: Based on data dependencies, under what scenarios and granularity can the identified loops act as a producer-consumer pair in a pipeline.)

Explain size choices for partial credit consideration.

Loop Pair	Size (bytes)
B→C	1 or 4
C→D	4
D→F	4096

B→C: Each byte read can be passed directly to C, and C can perform a lookup. Technically, C may read a whole 32b word. However, depending on length, it may consume less than a byte on each iteration. If C is consuming less than a byte, it can use each byte as it shows up. If C is consuming a whole 32b word, then it will need to get a full word (4 bytes) to be able to perform each operation.

C→D: As each fa[i] is produced, C can pass it to D, allowing D to perform one loop body on that fa[i]. fa is produced by C and consumed by D in order.

D→F: ts[] is updated on every invocation of D. The final value of ts[] is not known until the D completes the final iteration. As such, D cannot pass ts[] to F until it completes its execution. Then the whole ts[] (WINDOW×4=4096 bytes) can be given to F. F can write ts[] out while D is operating on the next iteration of the outer while loop.

So, the whole B→C→D body can operate as a pipeline. B and C can operate on data in the same outer while iteration, passing data in bytes or words as they are produced, while F must operate on data from an earlier outer while iteration than B and C.

6. Consider trying to achieve a real-time rate of one window output per cycle (equivalently, the II of the outer while loop is WINDOW or 1024 cycles).
   Assume you exploit data streaming between loops so they can run concurrently.

   (a) Given that Flash memory has a maximum throughput of 100 MB/s, what is the maximum possible value for FREQBYTES?
   100MB/s throughput, means the fastest we can read each byte is once ever 10 cycles.
   FREQBYTES $\times$ 10 = 1024 $\rightarrow$ FREQBYTES=102.

   (b) Based on your II identified in Problem 2c, what is the maximum value for `freqs` in order fo meet this real-time throughput goal?
   freqs $\times$ II = 1024 $\rightarrow$ freqs $\times$ 5 = 1024 $\rightarrow$ freqs=204.

   (c) What II do you need to achieve for Loop D to meet this real-time throughput goal?
   The most direct argument is that this needs to match the rate of Loop C, so also has an II=5 requirement.
   Alternately, we have the same equation, now with II_D as the variable.
   freqs $\times II_D$=1024 $\rightarrow$ 204$\times II_D$=1024 $\rightarrow$ $II_D = 5$.

7. Define the composition of a custom VLIW datapath for loop C that can achieve the identified II in Problem 2c.
   For full credit, minimize area of your implementation.
   Assume:

   - Design includes at least one write port to a scratchpad memory containing fa[] and one read port to a scratchpad memory containing in[]
   - Assume a crossbar interconnect between operator (and memory port) outputs and operator (and memory address, data) inputs.

   (a) How many operators of each type? Give both Resource Bound (RB) and number for which you can schedule.

Operator	Inputs	Outputs	**Number** **RB**	**Schedule**
shifters	2	1	$\lceil \frac{5}{5} \rceil = 1$	2
ALU (includes |, &, +, - , %-by-powers-of-2)	2	1	$\lceil \frac{7}{5} \rceil = 2$	2
scratchpad memory banks	2	1	1	1
ports to memory containing in[]	1	1	1	1
ports to memory containing fa[]	1 2	0 0	1	1
branch units	1	0	1	1

   above error, should be

   (b) How are the scratchpad memory banks used?
   Hold lookup[] array.

   (c) Crossbar Inputs and Outputs for your design (final column, the one you can schedule)?

Inputs	13 (or 14 with correction)
Outputs	6

(d) Estimate the area for your design using the following costs.

- shifters: 1024
- ALU (includes |, &, +, - , %-by-powers-of-2): 32
- Scratchpad memory banks of depth d: $60(d+6)$
- ports to memory containing in[]: 200
- ports to memory containing fa[]: 200
- branch unit: 100
- crossbar: $128 \times Inputs \times Outputs + 2400 \times Outputs$
  (Each crossbar output includes a 4 word memory acting as a small register file for input to the associated operator or memory.)

$2 \times 1024 + 2 \times 32 + 60(2048 + 6) + 200 + 200 + 100 + 128 \times 13 \times 6 + 2400 \times 6 = 150,236 \approx 150,000$

(e) Provide a schedule:

Operator→ / Cycle	fa[] write	in[] read	Label with your selected operators: lookup[]	shift0	shift1	ALU0	ALU1	Branch
0			res	t5				
1				tfa		len	next11	
2	fa[i]			t1		t2	next11bitpos	
3				t3	bytepos	bitoffset		
4		wordval				top11	t4	branch i<freqs
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								

Label cells with the variable assigned by the operation (or array entry written).

orange instructions software pipelined from previous iteration of loop

(Note extra schedules at end. May want to use as scratch while exploring schedules and put final here.)

8. Considering a custom hardware accelerator implementation where you are designing both the compute operators and the associated memory architecture, how would you use loop unrolling and array partitioning on Loop D to achieve the identified II in Problem 6c, while minimizing area?
Use the following area model and assume s[], ts[], and fa[] are part of this loop module:

   - n-bit counters: n
   - 32b adder: 32
   - 16×16 multiplier: 256
   - Single-port, 32b-wide memory holding d words: $38(d+6)$
   - Double-port, 32b-wide memory holding d words: $60(d+6)$

   (a) Unrolling for each loop (D, E)?

Loop	Unroll Factor
D	1
E	205

   To meet the $II = 5$ goal, we must perform $\lceil \frac{1024}{5} \rceil = 205$ loop bodies of E on each cycle. So, we can unroll E 205 times and pipeline the computation.

   (b) For the unrolling, how many multipliers and adders?

Multipliers	205
Adders	205

   Note: Since E is inside D, unrolling D D_{unroll} times and E E_{unroll} times, will result in $D_{unroll} \times E_{unroll}$ adders and multipliers.

   (c) Array partitioning for each array (s[], ts[], fa[])?
   (each memory block has either 1 or 2 ports)

Array	Array Partition	Ports (select one)		Words/partition
s[][]	cyclic 205 dimension 1	(1)	2	1280
ts[]	cyclic 205	1	(2)	5
fa[]	1	1	(2)	256

   Note that s[] is only read. ts[] must be both read and written on each iteration. fa[] must be written by C and read by D.

Properly pipelined fa[] could get away with one port; when C and D are run concurrently fa[] could go away as a memory and just become a register between C and D.

(d) Identify the component(s) that consumes most (>80%) of the area?
(you don't necessarily need to compute the area to fine precision, but you need to estimate where area is going well enough to answer the question above.)

Component	Calculate	Area
8-bit counter for E 3-bit counter for D	8 3	11
Adder	32×205	6560
Multiplier	256×205	52480
s[][]	$205 \times 38(1280 + 6)$	10017940
ts[]	$205 \times 60(5 + 6)$	135300
fa[]	$60(256 + 6)$	15720
total		10228011

98% of area is the single-ported memory for s[][].

Memory (for s[][]) consumes >80% of the area.

Extra schedule (in case you need it for trying schedules out, or if you need to put your answer here; be clear which schedule we should grade.)

Operator→ / Cycle	fa[] write	in[] read	Label with your selected operators										
0													
1													
2													
3													
4													
5													
6													
7													
8													
9													
10													
11													
12													
13													
14													

Label cells with the variable assigned by the operation (or array entry written).

Extra schedule (in case you need it for trying schedules out, or if you need to put your answer here; be clear which schedule we should grade.)

Operator→ / Cycle	fa[] write	in[] read	Label with your selected operators										
0													
1													
2													
3													
4													
5													
6													
7													
8													
9													
10													
11													
12													
13													
14													

Label cells with the variable assigned by the operation (or array entry written).

Consider the following code to render augmented reality features on a real-time video stream

code_one.c Fri Dec 20 10:14:13 2019 1

```
int WIDTH 4096
int HEIGHT 2048
int COLORS 3
int MASK 3

int VPARAMS 5
int VP_X 0
int VP_Y 1
int VP_XS 2
int VP_YS 3
int VP_ROT 4

int XOFF 1
int YOFF 1
int ROT 1
int XSCALE 2
int XFACT 2 // typo for XSFACT in original
int XSFACT 2
int YSCALE 2
int YSFACT 2 // typo for YSFACT in original
int YFACT 2

uint16_t reference[HEIGHT][WIDTH][COLORS];
uint16_t overlay[HEIGHT][WIDTH][COLORS+1]; // +1 for mask
int16_t sintable[360]; // -1 to 1 -- scaled by 2^14
int16_t costable[360];

void main() {
  while (true) { // loop Z
    augment_frame();
  }
}

void augment_frame() {
  uint16_t raw[HEIGHT][WIDTH][COLORS]; // uint16_t for 16b (2 byte) color per pixel
  uint16_t augment[HEIGHT][WIDTH][COLORS];
  uint16_t augmented[HEIGHT][WIDTH][COLORS];
  uint16_t old_viewpoint[VPARAMS];
  uint16_t viewpoint[VPARAMS];
  uint16_t *tmp_viewpoint;
  get_image(raw);
  tmp_viewpoint=old_viewpoint;
  old_viewpoint=viewpoint;
  viewpoint=tmp_viewpoint;
  compute_viewpoint(raw,reference,old_viewpoint,viewpoint);
  render_augmentation(viewpoint,overlay,augment);
  merge_frames(reference,viewpoint,raw,augment,augmented);
  send_image(augmented);
}
```

code_two.c Mon Dec 23 18:18:34 2019 1

```
void compute_viewpoint(uint16_t ***image, uint16_t ***reference,
                       int16_t  *old, int16_t  *current)
{
  uint64_t best_score=MAXINT; // maximum representable integer

  for (int rot=old[VP_ROT]-ROT;rot<old[VP_ROT]+ROT;rot+=1) { // loop A
    int16_t sr=sintable[rot]; // result is a fraction
    int16_t cr=costable[rot];
    for (int x=old[VP_X]-XOFF;x<old[VP_X]+XOFF;x++) // loop B
      for (int y=old[VP_Y]-YOFF;y<old[VP_Y]+YOFF;y++) // loop C
        for (int xs=old[VP_XS]/XSCALE;xs<old[VP_XS]*XSCALE;xs*=XSFACT) // loop D
          for (int ys=old[VP_YS]/YSCALE;ys<old[VP_YS]*YSCALE;ys*=YSFACT) // loop E
            {
              uint64_t score=0;
              for (int iy=0;i<HEIGHT;iy++) // loop F
                for (int ix=0;i<WIDTH;ix++) // loop G
                  {
                    uint16_t tx=((ix*cr+iy*sr)*xs)>>(14+8)+x; // 14 to scale sr, cr
                    uint16_t ty=((ix*sr+iy*cr)*ys)>>(14+8)+y; // +8 for xscale, yscal
                    if ((tx>=0) && (tx<WIDTH) && (ty>=0) && (ty<HEIGHT))
                      for (int c=0;c<COLORS;c++) // loop H
                        score+=abs(image[iy][ix][c]-reference[ty][tx][c]);
                  }
              if (score<best_score)
                {
                  best_score=score;
                  current[VP_ROT]=rot;
                  current[VP_X]=x;
                  current[VP_Y]=y;
                  current[VP_XS]=xs;
                  current[VP_YS]=ys;
                }
            }
  }
}

void render_augmentation(int16_t  *current, uint16_t ***overlay, uint16_t ***image)
{
  uint16_t rot=current[VP_ROT];
  uint16_t x=current[VP_X];
  uint16_t y=current[VP_Y];
  uint16_t xs=current[VP_XS];
  uint16_t ys=current[VP_YS];
  int16_t sr=sintable[rot]; // result is a fraction
  int16_t cr=costable[rot];
  for (int iy=0;i<HEIGHT;iy++) // loop I
    for (int ix=0;i<WIDTH;ix++) // loop J
      image[iy][ix]=UNMAPPED; // assume this runs like streaming data copy
  for (int iy=0;i<HEIGHT;iy++) // loop K
    for (int ix=0;i<WIDTH;ix++) // loop L
        {
          uint16_t tx=((ix*cr+iy*sr)*xs)>>(14+8)+x; // 14 to scale sr, cr
          uint16_t ty=((ix*sr+iy*cr)*ys)>>(14+8)+y; // +8 for xscale, yscale
          if ((tx>=0) && (tx<WIDTH) && (ty>=0) && (ty<HEIGHT)
              && (overlay[ty][tx][MASK]>0))
               for (int c=0;c<COLORS;c++) // loop M
                  image[iy][ix][c]=overlay[ty][tx][c];
        }

}
```

code_three.c Thu Dec 19 05:26:56 2019 1

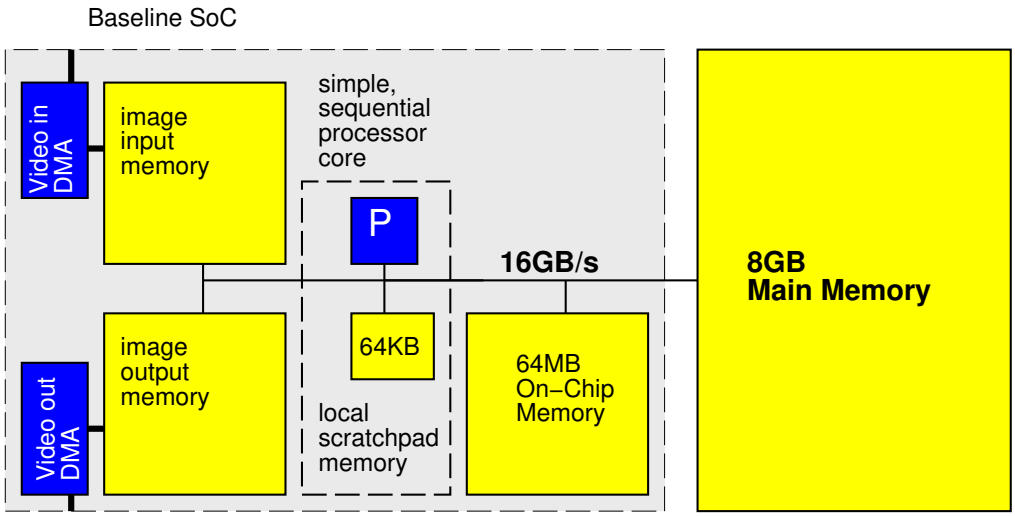
```
void merge_frames(uint16_t ***reference, int16_t  *current,
                  uint16_t ***image, uint16_t ***augment, uint16_t ***augmented)
{
  uint16_t rot=current[VP_ROT];
  uint16_t x=current[VP_X];
  uint16_t y=current[VP_Y];
  uint16_t xs=current[VP_XS];
  uint16_t ys=current[VP_YS];
  int16_t sr=sintable[rot]; // result is a fraction
  int16_t cr=costable[rot];
  for (int iy=0;i<HEIGHT;iy++) // loop N
      for (int ix=0;i<WIDTH;ix++) // loop O
        {
          uint16_t tx=((ix*cr+iy*sr)*xs)>>(14+8)+x; // 14 to scale sr, cr
          uint16_t ty=((ix*sr+iy*cr)*ys)>>(14+8)+y;// +8 for xscale, yscale
          if ((tx>=0) && (tx<WIDTH) && (ty>=0) && (ty<HEIGHT)
              && (augment[iy][ix]!=UNMAPPED))
            {
              uint32_t diff=0;
              for (int c=0;c<COLORS;c++) // loop P
                diff+=abs(image[iy][ix][c]-reference[ty][tx][c]);
              if (diff<THRESH)
                for (int c=0;c<COLORS;c++) augmented[iy][ix][c]=augment[iy][ix][c];
              else
                for (int c=0;c<COLORS;c++) augmented[iy][ix][c]=image[iy][ix][c];
            }
          else
            for (int c=0;c<COLORS;c++) augmented[iy][ix][c]=image[iy][ix][c];
        }
}

void get_image(uint16_t ***image)
{
  for (int iy=0;i<HEIGHT;iy++)
    for (int ix=0;i<WIDTH;ix++)
      for (int c=0;c<COLORS;c++)
        image[iy][ix][c]=image_in[iy][ix][c];
}

void send_image(uint16_t ***image)
{
  for (int iy=0;i<HEIGHT;iy++)
    for (int ix=0;i<WIDTH;ix++)
      for (int c=0;c<COLORS;c++)
        image_out[iy][ix][c]=image[iy][ix][c];

}
```

We start with a baseline, single processor system as shown.



- For simplicity throughout, we will treat non-memory indexing adds (subtracts count as adds), compares, abs, shifts, and multplies as the only compute operations. We'll assume the other operations take negligible time or can be run in parallel (ILP) with the adds, abs, shift, multiplies, and memory operations. (Some consequences: You may ignore loop and conditional overheads in processor runtime estimates; you may ignore computations in array indecies.)
- Baseline (simple, sequential) processor can execute one multiply, compare, shift, abs, or add per cycle and runs at 1 GHz.
- Data can be transfered between pairs of memory (including main memory) at 16 GB/s when streamed in chunks of at least 2048B. Assume `for` loops that only copy data can be auto converted into streaming operations.
- Non-streamed access to the main memory takes 100 cycles and can move 8B.
- Non-streamed access to image and 64 MB on-chip memories takes 10 cycles and can move 8B.
- Baseline processor has a local scratchpad memory that holds 64KB of data. Data can be streamed into the local scratchpad memory at 16 GB/s. Non-streamed accesses to the local scratchpad memory take 1 cycle.
- Baseline processor is 1 mm^2 of silicon including its 64KB local scratchpad.
- By default, all arrays live in the 8 GB main memory.
- `image_in` and `image_out` live in the respective image input and image output memories.
- Arrays for `sintable`, `costable` and viewpoints (`old_viewpoint`, `viewpoint`) live in local scratchpad memory.
- Assume scalar (non-array) variables can live in registers.
- Assume all additions are associative.
- Assume comparisons, adds, and multiplies take 1 ns when implemented in hardware accelerator, so fully pipelined accelerators also run at 1 GHz. A compare-mux operation can also be implemented in 1 ns. Consider abs and shift free in hardware.
- Data can be transfered to accelerator local memory at the same 16 GB/s when streamed in chunks of at least 2048B.

1. Simple, Single Processor Resource Bounds

   (a) Based only on the resource bound for compute operations, what throughput can a simple, single processor system achieve [answer in frames/second, or equivalently, `augment_frame` calls per second]?

get_image	all DMA	0
compute_viewpoint	$2^5 \times 4096 \times 2048 \times (12 + 3 \times 3)$ cycles	5.6 s
render_augmentation	$4096 \times 2048 \times 12$ cycles	0.10 s
merge_frames	$4096 \times 2048 \times (12 + 3 \times 3)$ cycles	0.18 s
send_image	all DMA	0
Total		5.9 s

0.17 frames/second

   (b) Based only on the resource bound for memory operations, what throughput can a simple, single processor system achieve [answer in frames/second]?

get_image	$\frac{4096 \times 2048 \times 3 \times 2}{16 \times 10^9}$	0.0031 s
compute_viewpoint	$2^5 \times 4096 \times 2048 \times (2 \times 100)$ cycles image[iy][ix] and reference[ty][tx] is single read ignore small terms sin/costable, current update	54 s
render_augmentation	$\frac{4096 \times 2048 \times 3 \times 2}{16 \times 10^9} +$ $4096 \times 2048 \times 2 \times 100$ cycles overlay[ty][tx] including mask is single read image[iy][ix] is single write	1.7s
merge_frames	$4096 \times 2048 \times 4 \times 100$ cycles	3.4 s
send_image	$\frac{4096 \times 2048 \times 3 \times 2}{16 \times 10^9}$	0.0031
Total		59 s

0.017 frames/second

2. Based on the simple, single processor mapping from Question 1:

   (a) What function is the bottleneck? (circle one)

get_image
(compute_viewpoint)
render_augmentation
merge_frames
send_image

   (b) What is the Amdahl's Law speedup if you only accelerate the identified function?
   $\frac{64.9}{5.4} = 11$

3. Data Parallel and Reduce: Classify Loops

Loop	Data Parallel?	Associative Reduce?	Must be Sequential?
A		X	
F		X	
K	X		
N	X		
Z (main)			X

Z must compute new viewpoint from one iteration/image before starting computation on next image.

A is a min-reduce on best_score.

F is a sum-reduce for score.

Computation for image[iy][ix] (K) and augmented[iy][ix] (N) are each independent of other elements of the respective arrays.

4. Data Streaming:

   (a) Can the producer and consumer operate concurrently on the same input image? or must the consumer work on a different (earlier) input image? ("Same Image?" column)

   (b) How big (minimum size) does the buffer (or other data storage space) need to be between the identified loops in order to allow the loops to profitably execute concurrently?

   (Hint: Based on data dependencies, under what scenarios and granularity can the identified loops act as a producer-consumer pair in a pipeline.)

Loop Pair	(a) Same Image?	(b) Size (bytes)
`get_image` → `compute_viewpoint`	N	48 MB
`compute_viewpoint`→`render_augmentation`	N	10 B
`render_augmentation` → `merge_frames`	Y	6 B
`merge_frames` → `send_image`	Y	6 B

Explain size choices for partial credit consideration.

Must hold onto an entire image from **`get_image`** to perform the search in **`compute_viewpoint`**.

Need to process entire search in **`compute_viewpoint`** before have a new viewpoint ($5 \times 2\text{B} = 10\text{B}$) to pass to **`render_augmentation`**. **`render_augmentation`** needs the viewpoint to process any image pixels.

As **`render_augmentation`** completes a pixel ($3 \times 2\text{B} = 6\text{B}$), it is ready to use, in the same order, in **`merge_frames`**.

As **`merge_frames`** completes a pixel ($3 \times 2\text{B} = 6\text{B}$), it is ready to be sent by **`send_image`** in the same order produced.

5. Latency Bound

   (a) What is the critical path (latency bound) for the entire computation as captured in the `augment_frame` function?

`compute_viewpoint`	read sintable, costable	1
	multiply by sine, cos	1
	add sin/cos terms	1
	scale	1
	(shifts for free)	0
	add offset	1
	read image and reference	100
	subtract	1
	(abs for free)	0
	sum reduce	$\log_2(4096 \times 2048 \times 3) = 25$
	min reduce	$\log_2(2^5) = 5$
`render_augmentation`	read sintable, costable	1
	multiply by sine, cos	1
	add sin/cos terms	1
	scale	1
	(shifts for free)	0
	add offset	1
	read overlay	100
	(don't write image, just use it below)	0
`merge_frames`	compute tx, ty with above	0
	(reads, if needed, happen with overlay above)	0
	subtract	1
	(abs for free)	0
	sum reduce	$\log_2(3) = 2$
	(don't write image, just use for output)	0
Total		244

(b) What is the latency bound Iteration Internal (II) for the `main` computation? (Hint: builds on part (a).)

136

Only need to compute new viewpoint.

6. Consider rewriting the body of `compute_viewpoint` to minimize the memory resource bound by exploiting the scratchpad memory and the 64 MB on-chip memory and streaming data tranfers.

   (a) Identify new temporary arrays allocated to scratchpad memory or 64MB on-chip memory (and specify which memory each new array is in).

   ```
   uint16_t image_line[WIDTH][COLORS]; // scratchpad
   uint16_t ref_copy[HEIGHT][WIDTH][COLORS]; // in 64MB on-chip memory
   ```

   (b) Describe how you use these arrays.

   Copy reference image into 64MB on-chip memory at beginning of function and operate on it from there.

   Copy each line ($4096 \times 3 \times 2$B) into image_line in the body of F before starting G. All references to image[iy][ix] now go to image_line.

   Common Problem: `reference` is accessed randomly. A line buffer will not work for it.

   (c) Account for total memory usage in the local scratchpad.

   24KB in image_line; 1440B in sintable and costable; 20B in old and current. Less than 26KB

   (d) Estimate the new memory resource bound for your optimized `compute_viewpoint`.

$$\frac{4096\times2048\times3\times2}{16\times10^9} + 2^5 \times 2048 \times \frac{4096\times3\times2}{16\times10^9} + \frac{2^5\times4096\times2048\times(10+1)}{10^9}$$

3.1 seconds

7. Consider a multiprocessor design that included N copies of a vector processor with 16 vector lanes, each operating on 16b data. This is a single-issue vector processor that can either issue one vector or one scalar operation on each cycle. Assume the loops you identifed as data parallel or reduce operation in Question 3 are perfectly vectorizable. Each vector processor requires 2 mm^2 including a local 64KB data scratchpad.

   (a) Based on computational requirements alone, how many vector processors do you need to achieve a 30 frame per second frame rate? [for this problem, ignore memory and communication]
   82

   (b) Identify how the processors are used.
   Everything that takes significant time is data parallel or an associative reduce.

`compute_viewpoint`	81
`render_augmentation`	1
`merge_frames`	(shared)

8. Considering a custom hardware accelerator implementation for `compute_viewpoint` where you are designing both the compute operators and the associated memory architecture. How would you use loop unrolling and array partitioning to achieve guaranteed throughput of 30 frames per second of throughput while minimizing area? Use the following area model in units of mm^2:

   - n-bit counters: $n \times 10^{-5}$
   - n-bit adder: $n \times 10^{-5}$
   - 16×16 multiplier: 2.5×10^{-3}
   - p-port, w-bit wide memory holding d words: $w(1+p)(d+6) \times 10^{-7}$

   Make the (probably unreasonable) assumption that reads from these memories can be completed in one cycle.

   Start by assuming we unroll H; we need to understand how much unrolling of the rest of the loops is required. Since the loops are associative reduce, the inner loop can be pipelined to II=1. $\frac{2^5 \times 4096 \times 2048}{A \times 10^9} \leq \frac{1}{30}$, giving us A a little over 8. This suggests unrolling about a factor of 16 beyond H will be sufficient.

   Common Problem: Not accounting for the operations that can be pipelined.

   (a) Unrolling for each loop?

Loop	Unroll Factor
A	1
B	1
C	1
D	1
E	1
F	1
G	16
H	3

   (b) For the unrolling, how many multipliers and adders?

Multipliers	$6 \times 16 = 96$
Adders	$16 \times (4 + 3 \times 2) = 160$

   (note: for upcoming area calculation, you will need to break down adders by size.)

   64b: 3×16=48, 16b: $(3+4) \times 7 = 112$

(c) Array partitioning for each array?

Note: blank rows left for local arrays you may have added when optimizing memory in Question 6.

Array	Array Partition	Ports	Width	Depth/partition
old[]	none	1	16	10
current[]	none	1	16	10
sintable[]	none	1	16	360
costable[]	none	1	16	360
image[]	n/a			
reference[]	n/a			
image_line[]	cyclic 16 dim 1, x complete dim 2 (and pack), c	1	48	256
ref_tmp[]	none	16	48	8,388,608

Common Problem: `reference` needs ports rather than partitioning since it is accessed randomly.

(d) Estimate the area for the accelerator.

Resource	Count	Area/resource	Area
16×16 multipliers	96	2.5×10^{-3}	0.24
64b-adders	48	64×10^{-5}	0.03072
16b-adders	112	16×10^{-5}	0.01792
3b-counters	5	3×10^{-5}	15×10^{-5}
8b-counters	1	8×10^{-5}	8×10^{-5}
12b-counters	1	12×10^{-5}	12×10^{-5}
memory (1,16,10)	2	5.1×10^{-5}	1.0×10^{-4}
memory (1,16,360)	2	1.2×10^{-3}	2.4×10^{-3}
memory (1,48,256)	16	2.5×10^{-3}	0.0402
memory(16,48,8388608)	1	680	680
Total			680