UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

MIDTERM EXAMINATION (**Solution**), October 29, 2013
ECE454H1 - Computer Systems Programming
Open book, open note
No programmable electronics allowed
Examiner – D. Yuan

Please Read All Questions Carefully! Please keep your answers as concise as possible.
You do not need to fill the whole space provided for answers.

*There are **15** total numbered pages, **11** Questions.*
You have 2 hours. Budget your time carefully!

Please put your FULL NAME, UTORid, Student ID on THIS page only.

Name: _____

UTORid: _____

Student ID: _____

Grading Page

DO NOT WRITE ON THIS PAGE

	Total Marks	Marks Received
Question 1	6	
Question 2	8	
Question 3	6	
Question 4	6	
Question 5	7	
Question 6	10	
Question 7	10	
Question 8	8	
Question 9	9	
Question 10	14	
Question 11	16	
Total	100	

Question 1 (6 marks): Branch prediction

Modern processors are equipped with branch prediction logic. What are the consequences if a processor does not have branch prediction logic?

Answer: Without branch prediction, the pipeline will be stalled by a branch instruction. This will significantly degrade the performance of the system considering branch instructions are 15%-25% of all instructions.

Question 2 (8 marks): CPU performance

When we are comparing two CPUs:

A). Does faster clock cycle always yield better system performance? why? (4 marks)

Answer: No. Different CPU might have different IPC --- an instruction might take more cycles to complete on a CPU with higher clock cycle, and thus takes more time to execute on a cpu with higher clock cycle.

B). Does a higher IPC always yield better system performance? why? (4 marks)

Answer: No. Clock speed might be different, expressiveness of ISA might also be different.

Question 3 (6 marks): Measurement tools

Running the “time” command on Linux prints out the real elapsed time (real), CPU time in user space (user) and kernel space (kernel).

A). Running “time a.out” prints the following outputs:

```
real  0m13.860s
user  0m10.669s
sys   0m0.720s
```

Why user + sys can be less than real? (3 marks)

Answer: User + sys only measures CPU time. If the program is doing I/O, for example, user + sys will be less than real. Context switches might also occurred.

B). Running “time b.out” prints the following outputs:

```
real  0m3.515s
user  0m10.837s
sys   0m0.672s
```

Why in this case user + sys can be greater than real? (3 marks)

Answer: “b.out” is a multi-threaded program executed on multi-core machine. The CPU time of each thread will be added into user/sys.

Question 4 (6 marks): Speed-up

Assume memory operations currently take 40% of execution time on a system that does not have cache. If adding an L1 cache speeds up 80% of memory operations by a factor of 4, and an L2 cache speeds up 1/2 the remaining 20% by a factor or 2, what is the total speed-up?

Answer:

Speedup = oldtime/newtime

Assume oldtime = T

memory operations takes 0.4T, other code takes 0.6T

After cache: memory operations takes:

(0.8/4 + 0.1 / 2 + 0.1) * 0.4T = 0.35 * 0.4T = 0.14T

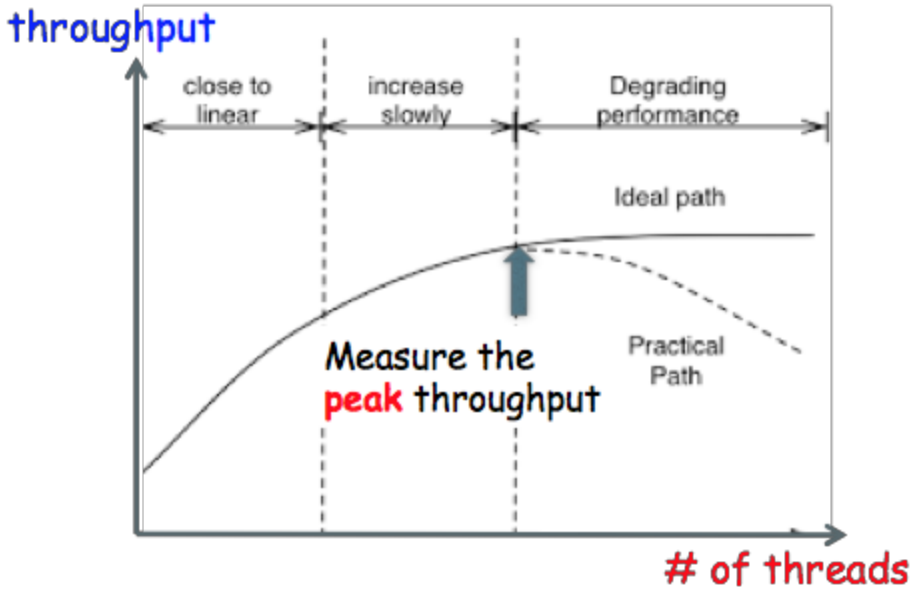
Newtime = 0.14T + 0.6T = 0.74T

Speedup = T/0.74T = 1.35

Question 5 (7 marks): Throughput vs. parallelism

"Throughput" is a commonly-used performance metric for servers. A common belief is that server throughput can be improved by increasing parallelism (e.g., multi-threads). Is it always true? Why or why not?

Answer: No. The curve between the # of threads and throughput is a follow. The reason for the decline is that if more threads are created than the system can handle, they will compete for system resources and the overhead of these threads will negatively impact the throughput.



Question 6 (10 marks): Memory alignment

A). Why the memory addresses are aligned by compilers? (4 marks)

Answer: to improve the system's memory access performance. In particular, CPU always read at WORD size of memory content; Cache line always starts at an address that is the multiple of cache line size. If the memory address for a variable is not aligned, it might take unnecessary number of memory accesses to read/write the variable.

Consider the following structure in C:

```
struct random_struct {
  int i; // 4 bytes
  char ch; // 1 byte
  double x; // 8 bytes
  char k; // 1 byte
  double y; // 8 bytes
  short z; // 2 bytes
} S[10];
```

B). How many bytes of memory will the array S occupy? (3 marks)

Answer:

```
| i(4B) | ch(1B) | 3B pad. | x(8B) | k(1B) | 7B pad. | y(8B) | z(2B) | 6B pad. |
^       ^        ^         ^       ^       ^         ^       ^       ^         ^
p      p+4      p+5       p+8     p+16    p+17      p+24    p+32    p+34      p+40

Each element needs 40 bytes. The array S takes 400 bytes.
```

C). Give a space-optimized version; how many bytes will it take? (3 marks)

```
struct random_struct {
  char ch; // 1 byte
  char k; // 1 byte
  short z; // 2 bytes
  int i; // 4 bytes
  double x; // 8 bytes
  double y; // 8 bytes
} S[10];
```

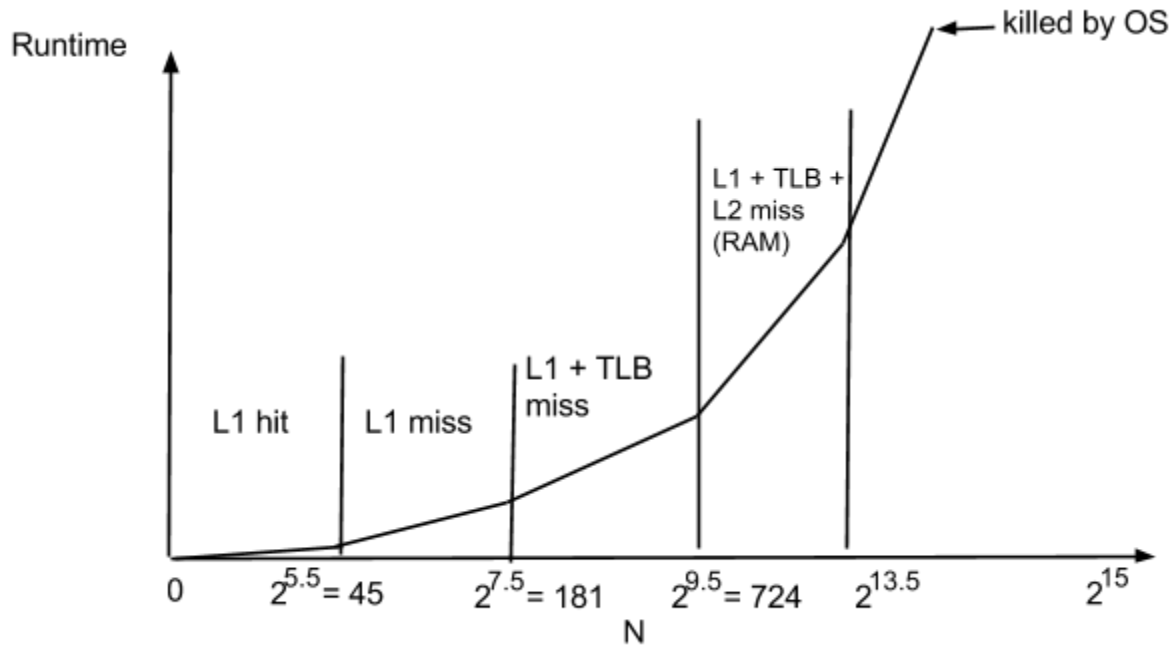
This version needs only 24 bytes per element, total 240 bytes. No padding is needed.

Question 7 (10 marks): Memory performance

Consider the following code fragment:

```
  /* a and b are N x N integer matrix (an integer has 32 bits) */
1 for (i = 0; i < N; i++)
2   for (j = 0; j < N; j++)
3     for (k = 0; k < N; k++)
4       sum += a[i][k] * b[k][j];
```

Assume a 32-bit architecture with L1 data cache size is 16KB, L2 cache size is 4MB, TLB has 64 entries and each page is of 4KB size, the RAM size is 1GB. Assume the entire data cache can be used to store a[] and b[] (other variables are all in register). Please draw a curve below to show how the runtime changes with the increase of N from 0 to 2^{15}. You do not have to draw the concrete values for the Y-coordinates, but you need to clearly mark the X-coordinates values for the key points in this graph. *Explain your answer.*



1. L1 size = 2^14B = 2^12 integers, so each array can at most have 2^11 integers to fit into L1. Therefore, when N <= square(2^11) = 45, L1 cache can hold all the data.

2. TLB reach = 64 * 4K = 256KB = 2^6 * 2^12 = 2^18B = 2^16 integers, so each array can at most have 2^15 integers before TLB starts to suffer misses. Therefore, when 45 < N <= square(2^15) = 181, there won't be TLB misses nor L2 misses, only L1 misses.

3. L2 size = 2^22B = 2^20 integers, so each array can at most have 2^19 integers before L2 starts to suffer misses. Therefore, when 181 < N <= square(2^19) = 724, there won’t be L2 misses, only L1 and TLB misses.

4. RAM size = 1GB = 2^30B = 2^28 integers, so each array can at most have 2^27 integers before OS starts swapping. Therefore, when 724 < N <= square(2^27) = 2^13.5, there won’t be swapping, only cache and TLB misses.

5. Finally, since the address space is 32-bits, we have 4GB = 2^32B address space = 2^30 integers, so each array can have 2^29 integers before running out of address space (in fact less than that since not the entire 4GB can be used by data). Therefore, before N reaches 2^15, we will run out of address space and OS will kill the process.

Grading: point 1-4 above worth 2 marks each. Another 2 marks for the general trend of the curve.

Question 8 (8 marks): 32-bit vs. 64-bit

A). When moving from 32-bit to 64-bit, what is the benefit? (3 marks)

Answer: the key benefit is that now the virtual address space for the process is increased from 2^32 = 4KB to 2^64B.

B). How do you write a simple C program to tell whether the underlying architecture is 32-bits or 64-bits? Note you cannot assume that the underlying OS explicitly expose the architecture information for you (i.e., you cannot read /proc/cpuinfo or files under /sys/devices/system/). (5 marks)

Answer: A couple ways to solve this:

1. int arch_bits = sizeof (int *) == 4 ? 32 : 64; // the size of any pointer will be 32 or 64 bits
2. The address returned by malloc: address returned by malloc is always double-word aligned. On 32-bit machine, it will always be a multiple of 8 (bytes); on 64-bit machine, should be 16 (bytes).

Question 9 (9 marks): Compiler optimizations

Consider the following functions:

```
int max(int x, int y) { return x < y ? y : x; }
void incr(int *xp, int v) { *xp += v; }
int add (int i, int j) { return i + j; }
```

The following code fragment calls these functions:

```
 1 int max_sum (int m, int n) { // m and n are large integers
 2   int i;
 3   int sum = 0;
 4
 5   for (i = 0; i < max(m, n); incr (&i, 1)) {
 6     sum = add(data[i], sum); // data is an integer array
 7   }
 8
 9   return sum;
10 }
```

Without considering loop-unrolling, answer Questions A) and B) below:

A). identify all of the optimization opportunities for this code and explain each one. Also discuss whether it can be performed by the compiler or not. (6 marks)

1. max(m,n) is loop invariant, therefore ILCM should move it out of the loop (compiler can do it).
2. max(m,n) can further be inlined (compiler can do it).
3. incr(&i, 1) can be inlined, simply replace by i++ --- compiler can do it.
4. add(data[i], sum) can be inlined (compiler can do it)

If answering compiler cannot perform optimizations and the explanation makes sense, no marks off.

B). Write down the optimized code (without loop-unrolling) (3 marks)

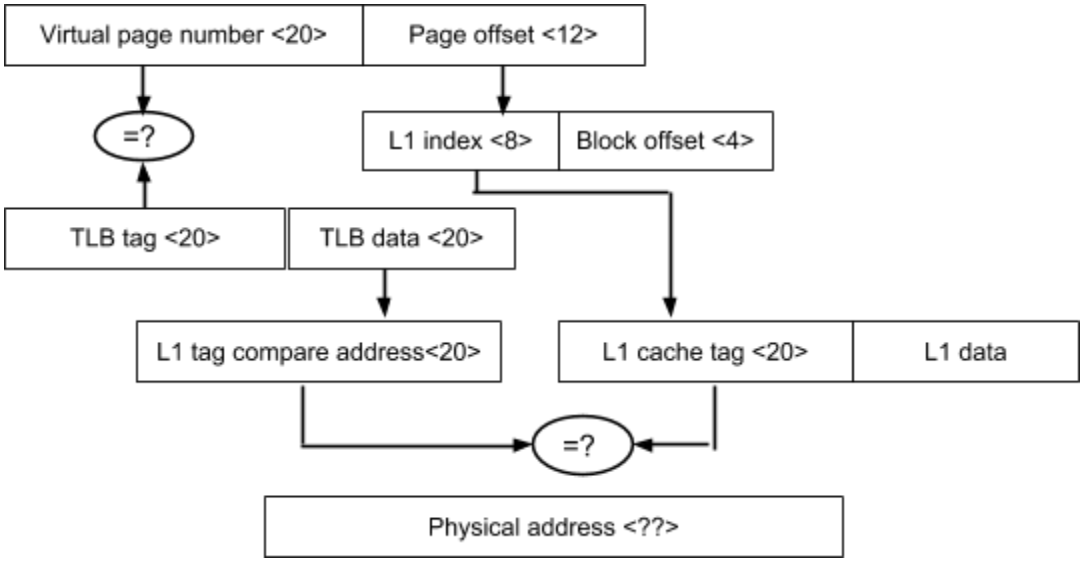
Answer:

```
int max_sum (int m, int n) { // m and n are large integers
   int i;
   int sum = 0;

   int tmp = m < n ? n : m;

   for (i = 0; i < tmp; i++) {
     sum = data[i] + sum;
   }
```

```
    return sum;
}
```

Question 10 (14 marks): Memory hierarchy

Consider the memory hierarchy pictured below:



The number in <> shows the number of bits for each component.

A). What is the size of a page? (2 marks)
Answer: 2^12 = 4KB.

B). How many sets are there in the L1 cache? (2 marks)
Answer: 2^8 = 256 sets.

C). Is the TLB fully associative? why? (2 marks)
Yes, because there are no index bits for TLB --- all the VPN will be used as tag comparison.

D). How many bits are there in Physical Address? (2 marks)
Answer: 32 -- 20 + 12.

E). How large is a cache line (or cache block)? (2 marks)
Answer: 2^4 = 16 bytes.

F). When the MMU receives a virtual address, how does it use each component in this diagram to read the content (assume TLB and L1 hit). (4 marks)
Answer: Given a virtual address (32 bits):

1st cycle: the first 20 bits will be used to perform TLB lookup. If this 20 bits matches with the TLB tag in one of the entries (and valid bit is 1), we have a TLB hit, and the TLB data will be the physical page number.

At the same time, the next 8 bits in the VA will be used to index into the L1 cache, taking us to the set A.

2nd cycle: we use the 20 bits from TLB data (physical page number) to perform tag comparison with the cache lines in A. If there is a match, and valid bit is 1, we have a L1 cache hit, and we further use the last 4 bits (block offset) to read the particular WORD from the L1 data and return to the CPU (may be in another cycle).

The cpu cycle analysis is not considered in marking.

Question 11 (16 marks): Loop unrolling (ding)

Suppose we wish to write a function to evaluate a polynomial, where a polynomial of degree n is defined to have a set of coefficients $a_0, a_1, a_2, \ldots, a_n$. For a value x, we evaluate the polynomial by computing

$$a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

This evaluation can be implemented by the following function, having as arguments an array of coefficients "a", a value "x", and the polynomial degree, "n". In this function, we compute both the successive terms of the equation and the successive powers of "x" within a single loop:

```
 1 int poly (int a[], int x, int n)
 2 {
 3   int i;
 4   int result = a[0];
 5   int xpwr = x; /* Equals x^i at start of loop */
 6   for(i = 1; i <= n; i++){
 7     result += a[i] * xpwr;
 8     xpwr = x * xpwr;
 9   }
10   return result;
11 }
```

Make the following assumptions of the underlying architecture:

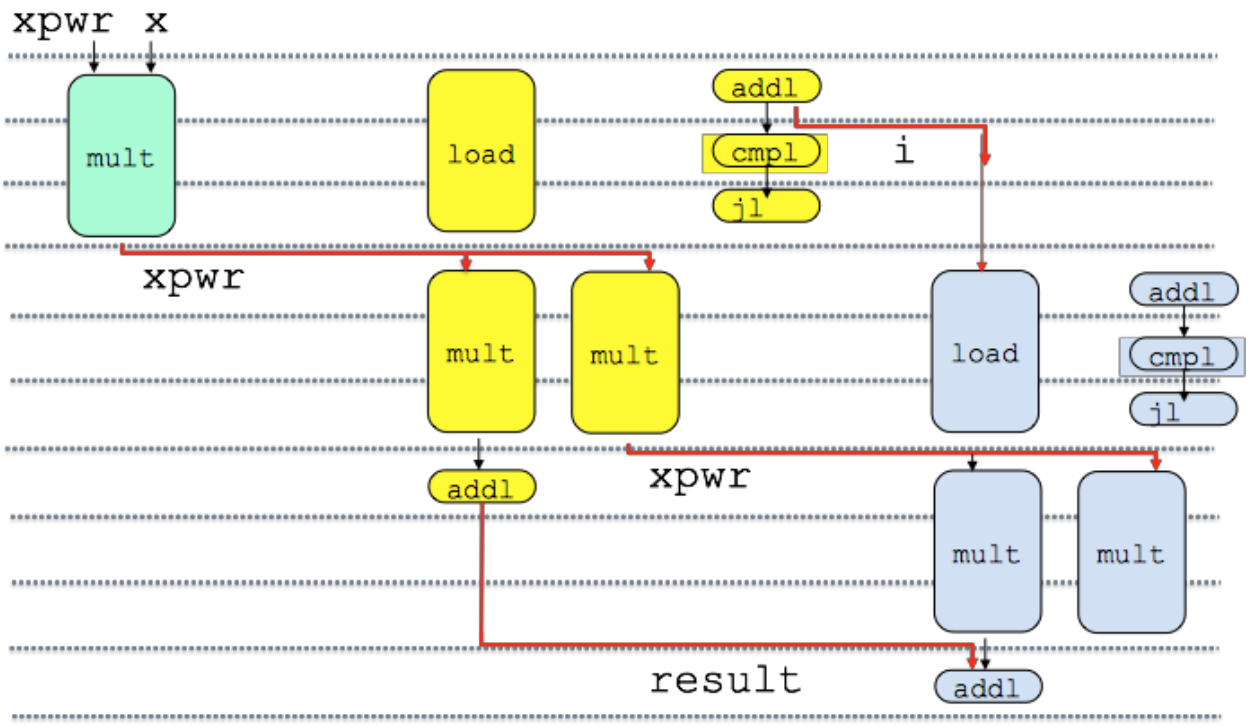
Operation	Latency (cycles)	Issue per cycle	Functional units
Addition (integer)	1	0.33	3
Multiplication (integer)	3	0.5	2
CMP (Compare)	1	1	1
Jump	1	1	1
Load	3	0.5	2

Note: "Issue per cycle" indicates the minimum number of cycles between two operations. For "Multiplication" and "Load", the machine has 2 functional units for each operation, and they're pipelined, therefore we can issue 2 multiplications and 2 loads every cycle even if each takes 3 cycles to finish. For Multiplication and Load, the first cycle will read in the input operands, and the result is written to the output register at the end of the 3rd cycle.

The machine is superscalar (can issue UNLIMITED number of different instructions each cycle), pipelined, has enough registers for any loop unrolling, and has accurate branch prediction.

You can make additional assumptions of the machine as long as they do not conflict with the above assumptions. In that case write down your assumptions.

A). Please identify all the data dependencies between two consecutive iterations of the loop between line 6 - 9. (4 marks)
Answer: see the graph below (data dependencies are marked as red):



1. both statements:

```
 7      result += a[i] * xpwr;
 8      xpwr = x * xpwr;
```

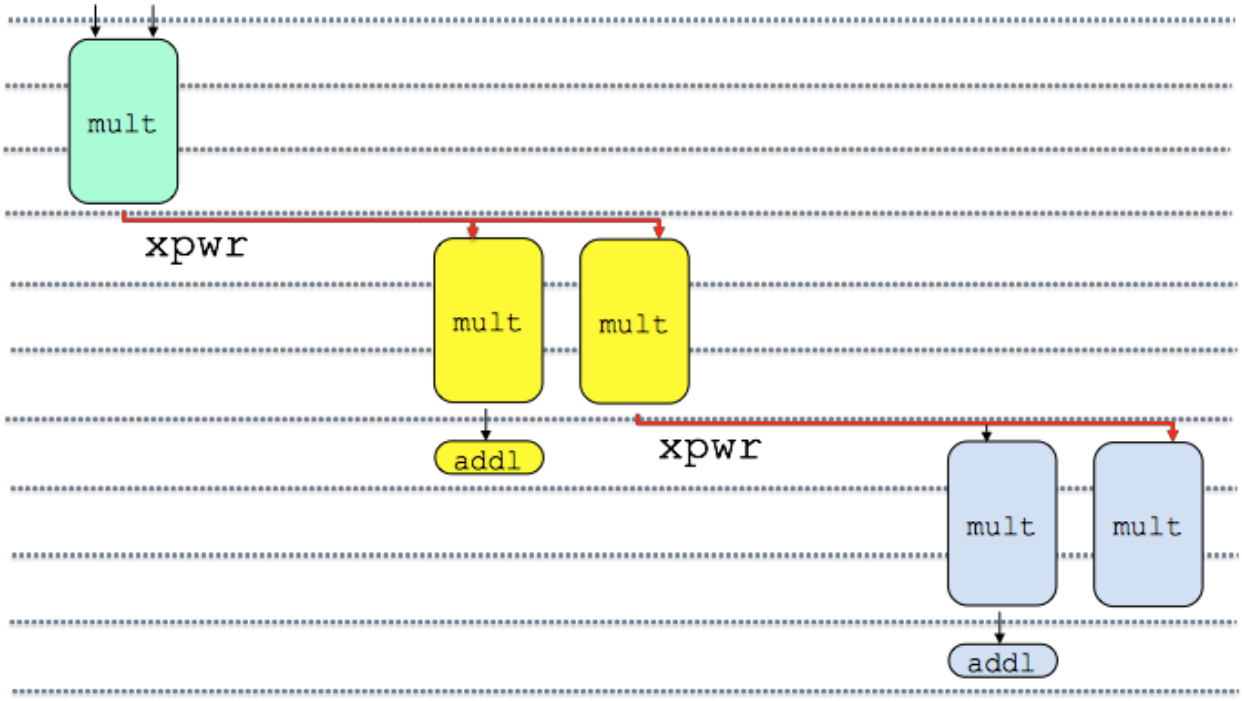
are dependent on the value “xpwr” from the previous iteration

2. result += a[i] * xpwr
The result is dependent on the same instruction in the previous iteration.

3. a[i] depends on the i++ from the previous iteration.

B). Based on the definitions from our lectures, what is the CPE for this function? Explain your answer based on the data dependencies formed between the loop iterations. Draw diagrams if necessary. (4 marks)

Answer: 3 cycles. See the critical path below (simplified from the graph above).



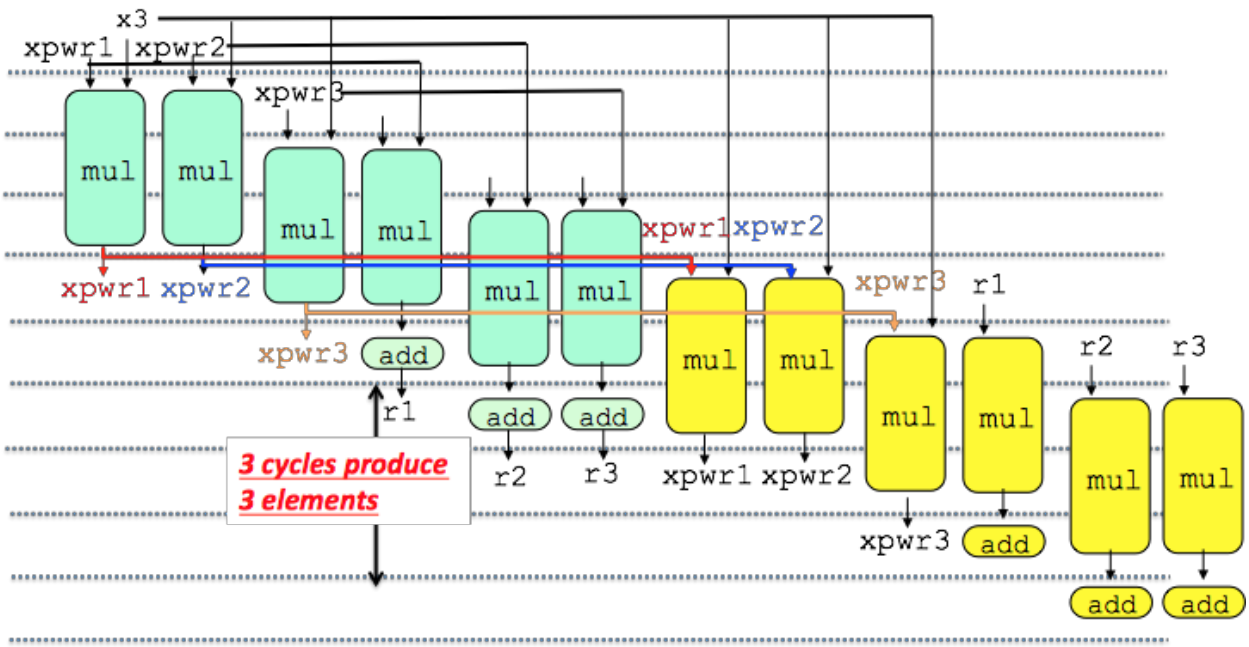
The critical path is in red (btw. the "`xpwr = x * xpwr;`" and the multiplications in the next iteration).

C). Can the CPE from B) be optimized by rewriting the code? If so, how, and what is the improved the CPE (explain your answer)? If not, why? (8 marks)

Answer: Yes. You can achieve a CPE = 1 by unrolling it 3 times. This is the best case without changing the algorithm ***(later we will discuss what if you can change the algorithm)***. You won't get any additional benefits by unrolling it more than 3 times. Unrolling it 2 times will give you a CPE = 1.5.

```
int poly_unroll3 (int a[], int x, int n)
 {
   int i;

   x3 = x * x * x;
   r1 = a[0];
   xpwr1 = x3;

   r2 = a[1] * x;
   xpwr2 = x * x3;

   r3 = a[2] * x * x;
   xpwr3 = x * xpwr2;

   len = n - 3;

   for(i = 3; i <= len; i+=3){
      r1 += a[i] * xpwr1;
      r2 += a[i+1] * xpwr2;
      r3 += a[i+2] * xpwr3;
      xpwr1 = x3 * xpwr1;
      xpwr2 = x3 * xpwr2;
      xpwr3 = x3 * xpwr3;
    }

    while (i <= n) {
      r1 += a[i++] * xpwr1;
      xpwr1 = x * xpwr1;
    }

    result = r1 + r2 + r3;
    return result;
}
```

The key here is that after the unrolling, Here is the diagram:

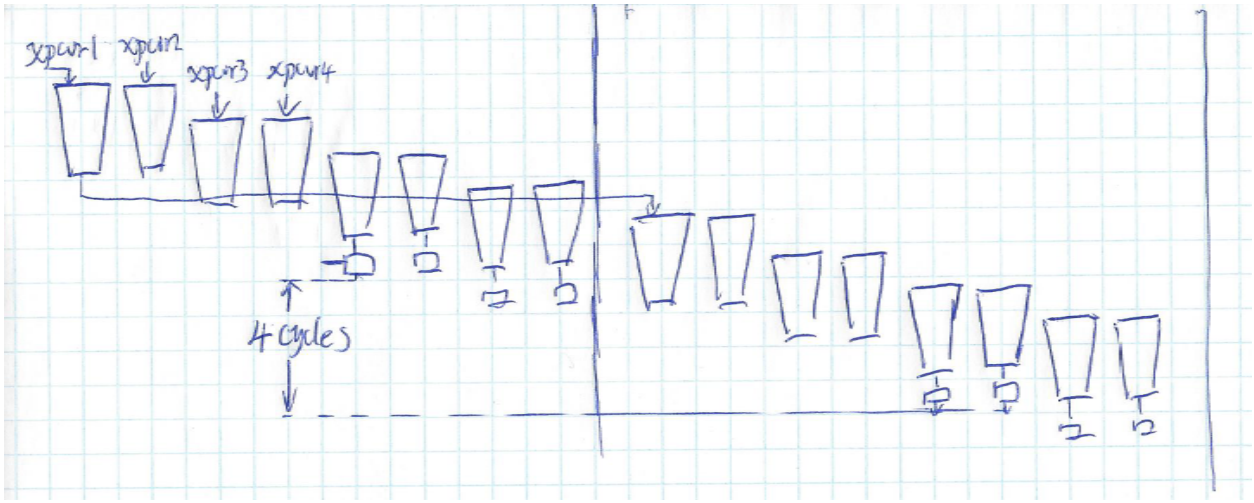


Every three cycles we can produce 3 elements, therefore the CPE is 1.
Here are some common mistakes we have seen:

1. "***unrolling cannot help because we only have 2 multipliers***". the key here is that the multipliers are pipelined! Therefore, as the graph above shows, while at any cycle we can only start 2 multiplications, the very next cycle we can start another 2 multiplications.

2. "xpwr1 in the above figure will be overwritten by the xpwr1 = xpwr1*x3 operation before it is used in a[i]*xpwr1 operation, therefore we cannot reorder the two multiplications and xpwr1 * x3 cannot start before a[i]*xpwr1 finishes."
Again, read the question more carefully, you will find out that we assume the multiplication will read in the input operand at the first cycle, and will write back the output register at the end of the 3rd cycle. Therefore even though we reordered the multiplication above, the xpwr1 used in the a[i] * xpwr1 is still the one before the xpwr1*x3 modifies it.

Below is the diagram showing what if we unroll it four times. Every 4 cycles it produces 4 elements, so it will give you the same CPE = 1.



Unrolling it five times won't form the perfect pipeline anymore, because the delay between the:

```
xpwr = x * xpwr
and
result += a[i]*xpwr
in the same iteration will be 3 cycles and we can no longer reorder them.
```

Now if we rewrite the algorithm, we can further reduce the CPE to 0.5. This solution is proposed by David Biancolin.

```
int poly_unroll8(int a[], int x, int n)
{
	int x2 = x * x;
	int x4 = x2 * x2;
	int x8 = x4 * x4;
	int i = 0;

	int sub_poly[8] = {0};

	//So the basic premise is the same as the original solution, however what we are
doing is essentially a parallel unroll of 8
	// p = a0 + a1x + a2*x^2 + ... + an*x^n ->

	// sp0 = a0 + a8*x^8 + a16*x^16 + ...
	// sp1 = a1*x + a9*x^9 + a17*x^17 + ...
	// sp2 = a2*x^2 + a10*x^10 + a18*x^18 + ...
	// ...
	// sp7 = a7*x^7 + a15*x^15 + a23*x^23 + ...

	//Observing that we may rewrite any of these sub polynomials as
```

```
	// spk = x^k*(ak + x^8(ak+8 + x^8(ak+16 + x^8(ak+24 + ...
	//           | iter z | iter z-1 |iter z-2   | iter z-3 ...

	// We can reduce the computation of a single polynomial to one multiply per element.
	// By multiplying the previous result, by x8, and adding the next smallest coefficient.


	//Some initialization. Since we may not be given a polynomial of degree 8*k-1 (where k is an integer)
	//we pop off the top l coefficients, such that ((n - l) + 1) mod 8 = 0

	// Find the greatest multiple of 8 < n, and initialize the coefficients
	const int largest_multiple_of_eight = (n&~7);
	for (i = largest_multiple_of_eight; i <=n i++) {
		sub_poly[i%8] = c[i];
	}

	for (i = largest_multiple_of_8-8; i>=0; i-=8)
	{

		//Implement spk = An + x^8(spk)
		//Note this has a latency of 4 cycles. But only uses 1 multiply per element.
		//With two muliply units, and two load units, we can therefore process two elements per cycle.
		//Reducing CPE from 1 to 0.5.

		//There may be register pressure depending on the ISA this is being compiled for. However,
		//even still, this should still outperform the 1 CPE solution.
		subpoly[0] *= x8;
		subpoly[0] += a[i+0];
		subpoly[1] *= x8;
		subpoly[1] += a[i+1];
		subpoly[2] *= x8;
		subpoly[2] += a[i+2];
		subpoly[3] *= x8;
		subpoly[3] += a[i+3];
		subpoly[4] *= x8;
		subpoly[4] += a[i+4];
		subpoly[5] *= x8;
		subpoly[5] += a[i+5];
		subpoly[6] *= x8;
		subpoly[6] += a[i+6];
		subpoly[7] *= x8;
		subpoly[7] += a[i+7];
```

```
	}
	//At this point we have 8 sub polynomials that look like

	//a0 + a8*x^8 + a16*x^16 + ...		-> need to multiply by 0
	//a1 + a9*x^8 + a17*x^16 + ...		-> need to multiple by x
	// ...
	//a7 + a15*x^8 + a23*x^16+ ...		-> need to multiply by x^7

	//Multiply each sub polynomial by their multiplicative offset and combine
	//to create result.

	int result = 0;
	int xpwr = x;
	for (i = 1; i < 8; i++){
		result +=xpwr*subpoly[i];
		xpwr *=x;
	}

	return result;
}
```