# CSPB 2400 - Park - Computer Systems

| | |
|---|---|
| **Started on** | Friday, 1 March 2024, 7:58 PM |
| **State** | Finished |
| **Completed on** | Friday, 1 March 2024, 8:29 PM |
| **Time taken** | 31 mins 21 secs |
| **Marks** | 11.00/12.00 |
| **Grade** | **9.17** out of 10.00 (**92**%) |

Question **1**

Correct

Mark 2.00 out of 2.00

Assume you have the following code

```
/* Accumulate in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
  long int i;
  int length = vec_length(u);
  data_t *udata = get_vec_start(u);
  data_t *vdata = get_vec_start(v);
  data_t sum = (data_t) 0;
  for (i = 0; i < length; i++) {
     sum = sum + udata[i] * vdata[i];
  }
 *dest = sum;
}
```

and you modify the code to use 4-way loop unrolling and four parallel accumulators. Measurements for this function with the x86-64 architecture shows it achieves a CPE of 2.0 for all types of data.

Assuming the model of the Intel i7 architecture shown in class (one branch unit, two arithmetic units, one load and one store unit), the performance of this loop with any arithmetic operation can not get below 2.0 CPE because of the number of available load units ✓ .

When the same 4x4 code is compiled for the IA32 architecture, it achieves a CPE of 2.75, worse than the CPE of 2.25 achieved with just four-way unrolling. The mostly likely reason this occurs is because not enough available registers ✓ .

Question **2**

Correct

Mark 1.00 out of 1.00

Assume you have the following code

```
/* Accumulate in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
  long int i;
  int length = vec_length(u);
  data_t *udata = get_vec_start(u);
  data_t *vdata = get_vec_start(v);
  data_t sum = (data_t) 0;
  for (i=0; i < length; i++) {
     sum = sum + udata[i] * vdata[i];
  }
 *dest = sum;
}
```

and you modify the code to the form below.

```
/* Accumulate in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
  long int i;
  int length = vec_length(u);
  data_t *udata = get_vec_start(u);
  data_t *vdata = get_vec_start(v);
  data_t sum = (data_t) 0;
  data_t sum2 = (data_t) 0;
  for (i = 0; i < length-1; i += 2) {
     sum = sum + udata[i] * vdata[i];
     sum2 = sum2 + udata[i+1] * vdata[i+1];
  }
  for (; i < length; i++) {
     sum = sum + udata[i] * vdata[i];
  }
 *dest = sum + sum2;
}
```

What type of optimizations is being applied?

Pick all optimizations the programmer performed that play a significant role in performance.

Select one or more:

- a. Strength reduction
- b. Common subexpression elimination
- [✓ c. Parallel accumulators] ✓
- d. Machine independent optimization
- [✓ e. Loop unrolling] ✓ Correct!
- f. Inlining
- g. Instruction pipelining

Your answer is correct.

This is an example of loop unrolling and multiple accumulators.

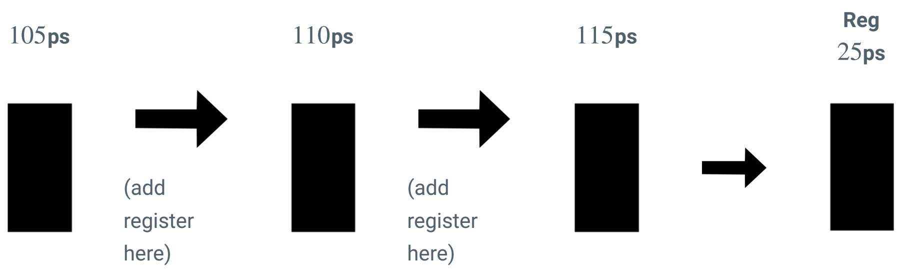
The correct answers are: Loop unrolling, Parallel accumulators

Question **3**

Correct

Mark 2.00 out of 2.00

We have a combinatorial logic function that can be decomposed into three steps each with the indicated delay with a resulting clock speed of 2.82 GHz.



Assume we further pipeline this logic by adding two additional registers. What would be the resulting clock speed in GHz?

7.142857143

You can use an expression if you like.

Your last answer was interpreted as follows: $7.142857143$

Correct answer, well done.
The cycle time of the fully pipelined design would depend on the largest stage delay ( $115$ ) and one register delay. This results in a total cycle time of $140$ with a resulting clock speed of $7.14285714286$ GHz.

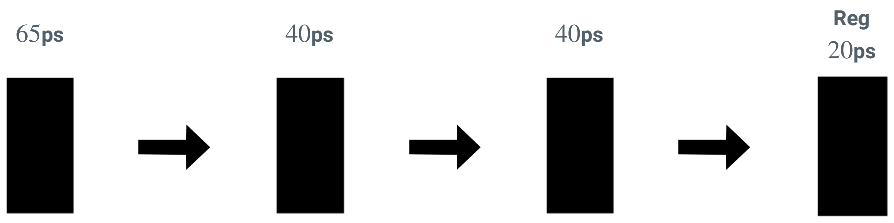
A correct answer is $\frac{50}{7}$, which can be typed in as follows: `50/7`

Question **4**

Incorrect

Mark 0.00 out of 1.00

We have a combinatorial logic function that can be decomposed into three steps each with the indicated delay with a resulting clock speed of $6.06$ GHz.



Assume we further pipeline this logic by adding just one additional register between the first two or last two stages of combinatorial logic.

What would be the highest resulting clock speed we could achieve in GHz? 8.333333333

You can use an expression if you like.

Your last answer was interpreted as follows: $8.333333333$

Incorrect answer.
Your answer $8.333333333$ is not close enough to the correct answer.

If we add the additional register between the first two logic blocks, the delays would be $65$ and $80$. If we put the register between the 2nd and 3rd block, the delays would be $105$ and $40$. The best overall delay is thus $80$, and the total pipeline delay is $80+20$. This results in a clock speed of $10.0$ GHz.

A correct answer is $10.0$, which can be typed in as follows: `10.0`

Question **5**

Correct

Mark 1.00 out of 1.00

The following code fragments each compute the same results. Which would incur the least delay due to pipelining on a computer with a single functional unit? You can assume all memory loads take the same time.

Select one:

a.
```
leaq 8(%rbx), %rdi
leaq 16(%ebc), %rbi
movq (%rdi), %rsi
sar   $4,%rsi
movq (%rbi),%rax
sal    $2,%rax
addq %rsi,%rax
```

[✓ b.]
```
leaq 8(%rbx), %rdi
leaq 16(%ebc), %rbi
movq (%rdi), %rsi
movq (%rbi),%rax
sar   $4,%rsi
sal    $2,%rax
addq %rsi,%rax
```
✓

c.
```
leaq 8(%rbx), %rdi
leaq (%rdi), %rsi
sar   $4,%rsi
movq 16(%ebc), %rbi
movq (%rbi),%rax
sal    $2,%rax
addq %rsi,%rax
```

Your answer is correct.

If two instructions of the same type are executed one immediately after another, for a single functional unit, they will be pipelined faster.

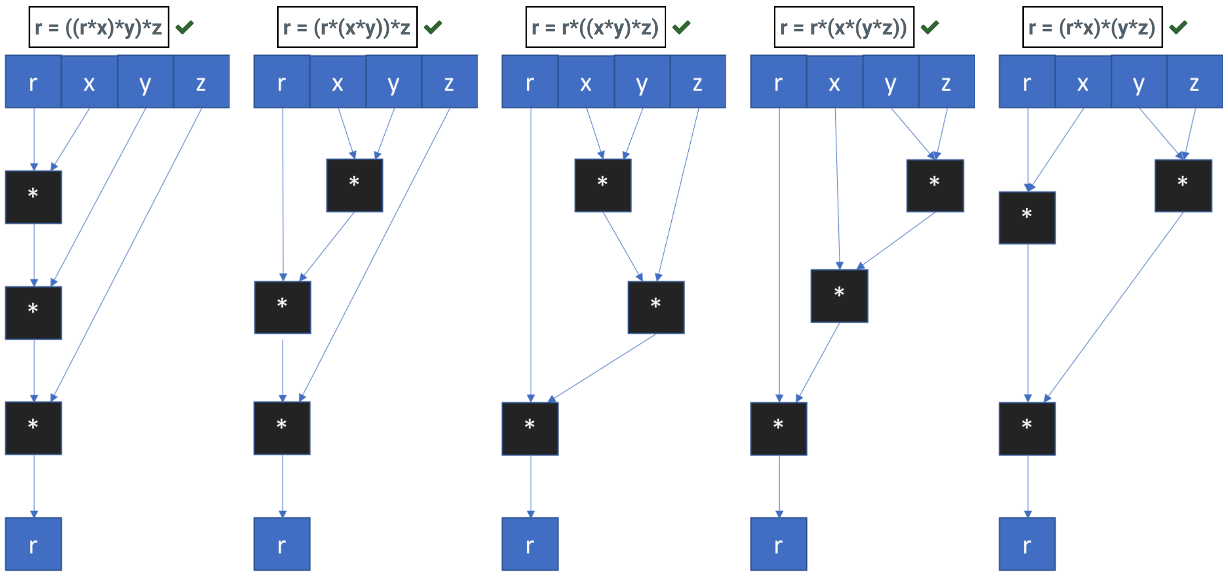
The correct answer is:
```
leaq 8(%rbx), %rdi
leaq 16(%ebc), %rbi
movq (%rdi), %rsi
movq (%rbi),%rax
sar   $4,%rsi
sal    $2,%rax
addq %rsi,%rax
```

Question **6**

Correct

Mark 5.00 out of 5.00

The following graphs illustrate different data flow graphs for the equations shown below. Drag the appropriate equation to the matching data flow graph that illustrates the order of evaluation.



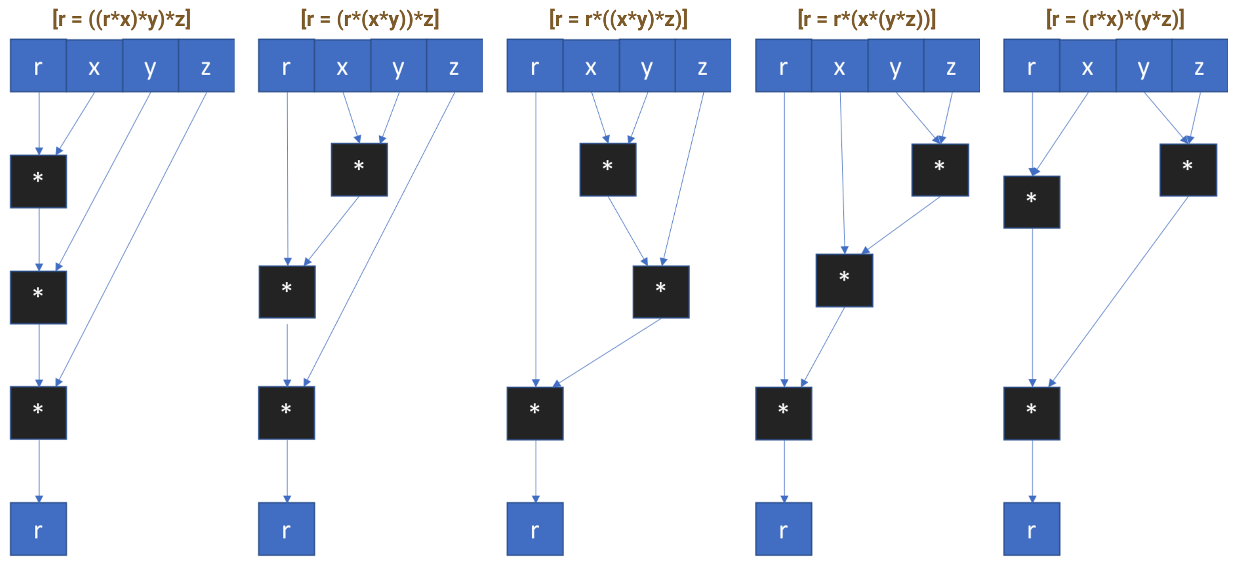
Your answer is correct.

For each of the figure it is very clear from the dependency.
Ex: [1] r*x should be computed first as the result is required to compute (r*x)*y and this is dependent on computing the next result ((r*x)*y)*z
Hence r should be r= ((r*x)*y)*z

Similar explanation holds for rest of expressions given.

The correct answer is:
The following graphs illustrate different data flow graphs for the equations shown below. Drag the appropriate equation to the matching data flow graph that illustrates the order of evaluation.

**[r = ((r*x)*y)*z]** **[r = (r*(x*y))*z]** **[r = r*((x*y)*z)]** **[r = r*(x*(y*z))]** **[r = (r*x)*(y*z)]**


[r = ((r*x)*y)*z]
[r = (r*(x*y))*z]
[r = r*((x*y)*z)]
[r = r*(x*(y*z))]
[r = (r*x)*(y*z)]
r
x
y
z
*
r