Universidad Carlos III de Madrid

Department of Computer Science and Engineering
Bachelor in Computer Science and Engineering
Computer Architecture

Exam

ATTENTION:

- **Time limit is 120 minutes.**

NAME:
FAMILY NAME:
NIA:

Exercise 1 [3 points]:

In a given processor we want to run the following code fragment:

```
loop:  l.d $f0, 0($r1)
       l.d $f2, 0($r2)
       l.d $f4, 0($r3)
       mul.d $f0, $f0, $f2
       add.d $f0, $f0, $f4
       s.d $f0, 0($r4)
       addi $r1, $r1, 8
       addi $r2, $r2, 8
       addi $r3, $r3, 8
       addi $r4, $r4, 8
       bne $r1, $r0, loop
```

In that processor latencies among instructions are:

Instruction producing result	Instruction using result	Latency
FP ALU operation	FP ALU operation	8
FP ALU operation	Store double	5
Load double	FP ALU operation	3
REST		0

Assume that the loop is executed a number of iterations that is a multiple of 8.

You are asked:

1. Determine the number of cycles required to run one iteration, if no modification to the code is made.
2. Determine the number of cycles required to run one iteration, when the loop has been scheduled.
3. Determine the number of cycles required to run one iteration, when loop unrolling and loop scheduling have been applied for the cases with an unrolling factor of 2, 4 and 8.
4. Explain differences in obtained results.

Solution

1)

26 cycles are required, including identified stalls:

```
loop:   l.d $f0, 0($r1)
        l.d $f2, 0($r2)
        l.d $f4, 0($r3)
        <stall> x 2
        mul.d $f0, $f0, $f2
        <stall> x 8
        add.d $f0, $f0, $f4
        <stall> x 5
        s.d $f0, 0($r4)
        addi $r1, $r1, 4
        addi $r2, $r2, 4
        addi $r3, $r3, 4
        addi $r4, $r4, 4
        bne $r1, $r0, loop
```

2) Loop scheduling can reduce cycles per interation to 22.

```
loop:   l.d $f0, 0($r1)
        l.d $f2, 0($r2)
        l.d $f4, 0($r3)
        addi $r1, $r1, 4
        addi $r2, $r2, 4
        mul.d $f0, $f0, $f2
        addi $r3, $r3, 4
        addi $r4, $r4, 4
        <stall> x 6
        add.d $f0, $f0, $f4
        <stall> x 5
        s.d $f0, -4($r4)
        bne $r1, $r0, loop
```

3)

```
loop:   l.d $f0, 0($r1)
        l.d $f2, 0($r2)
        l.d $f4, 0($r3)
        l.d. $f6, 4($r0)
        l.d $f8, 4($r0)
        l.d $f10, 4($r0)
        mul.d $f0, $f0, $f2
        addi $r1, $r1, 4
        mul.d $f6, $f6, $f8
        addi $r2, $r2, 8
        addi $r3, $r3, 8
        addi $r4, $r4, 8
        <stall> x 3
        add.d $f0, $f0, $f4
        <stall>
        add.d $f6, $f6, $f10
        <stall> x 3
        s.d $f0, -8($r4)
        <stall>
        s.d $f6, -4($r4)
        bne $r1, $r0, loop
```

In total, 25 cycles are required to run two iterations. Consequently, 12.5 cycles per iteration are required.

For a loop unrolling factor of 4:

```
loop:   l.d $f0, 0($r1)
        l.d $f2, 0($r2)
        l.d $f4, 0($r3)
        l.d. $f6, 4($r1)
        l.d $f8, 4($r2)
        l.d $f10, 4($r3)
        l.d $f12, 8($r1)
        l.d $f14, 8($r2)
        l.d $f16, 8($r3)
        l.d. $18, 12($r1)
        l.d $f20, 12($r2)
        l.d $f22, 12($r3)
        mul.d $f0, $f0, $f2
        mul.d $f6, $f6, $f8
        mul.d $f12, $f12, $f14
```

```
mul.d $f18, $f18, $f20
addi $r1, $r1, 16
addi $r2, $r2, 16
addi $r3, $r3, 16
addi $r4, $r4, 16
<stall>
add.d $f0, $f0, $f4
add.d $f6, $f6, $f10
add.d $f12, $f12, $f16
add.d $f18, $f18, $f22
<stall> x 2
s.d $f0, -16($r4)
s.d $f6, -12($r4)
s.d $f12, -8($r4)
s.d $f18, -4($r4)
bne $r1, $r0, loop
```

In total, 32 cycles are required for four iterations, giving 8 cycles per iteration.

For a factor of 8, an excessive number of registers would be needed and it would not be feasible.

However, if we assume a register file sufficiently large, we have:

```
loop:   l.d $f0, 0($r1)
        l.d $f2, 0($r2)
        l.d $f4, 0($r3)
        l.d. $f6, 4($r1)
        l.d $f8, 4($r2)
        l.d $f10, 4($r3)
        l.d $f12, 8($r1)
        l.d $f14, 8($r2)
        l.d $f16, 8($r3)
        l.d. $18, 12($r1)
        l.d $f20, 12($r2)
        l.d $f22, 12($r3)
        l.d $f24, 16($r1)
        l.d $f26, 16 ($r2)
        l.d $f28, 16 ($r3)
        l.d. $f30, 20($r1)
        l.d $f32, 20 ($r2)
        l.d $f34, 20 ($r3)
        l.d $f36, 24($r1)
        l.d $f38, 24($r2)
```

```
l.d $f40, 24($r3)
l.d. $42, 28($r1)
l.d $f44, 28($r2)
l.d $f46, 28($r3)
mul.d $f0, $f0, $f2
mul.d $f6, $f6, $f8
mul.d $f12, $f12, $f14
mul.d $f18, $f18, $f20
mul.d $f24, $f24, $f26
mul.d $f30, $f30, $f32
mul.d $f36, $f36, $f38
mul.d $f42, $f42, $f44
addi $r1, $r1, 32
addi $r2, $r2, 32
addi $r3, $r3, 32
addi $r4, $r4, 32
add.d $f0, $f0, $f4
add.d $f6, $f6, $f10
add.d $f12, $f12, $f16
add.d $f18, $f18, $f22
add.d $f24, $f24, $f28
add.d $f30, $f30, $f34
add.d $f36, $f36, $f40
add.d $f42, $f42, $f46
s.d $f0, -32($r4)
s.d $f6, -28($r4)
s.d $f12, -24($r4)
s.d $f18, -20($r4)
s.d $f24, -16($r4)
s.d $f30, -12($r4)
s.d $f36, -8($r4)
s.d $f42, -4($r4)
bne $r1, $r0, bucle
```

In total, 53 cycles for eight iterations, which gives 6.625 cycles per iteration.

4)

We notice that when loop unrolling factor is increased, the number of cycles per iteration is decreased. This is due to the fact that we achieve a better stall hiding with productive work and, consequently, the processor is better used.

However, we must keep in mind the increasing pressure on the register file. Thus, a higher loop unrolling may be excessive.

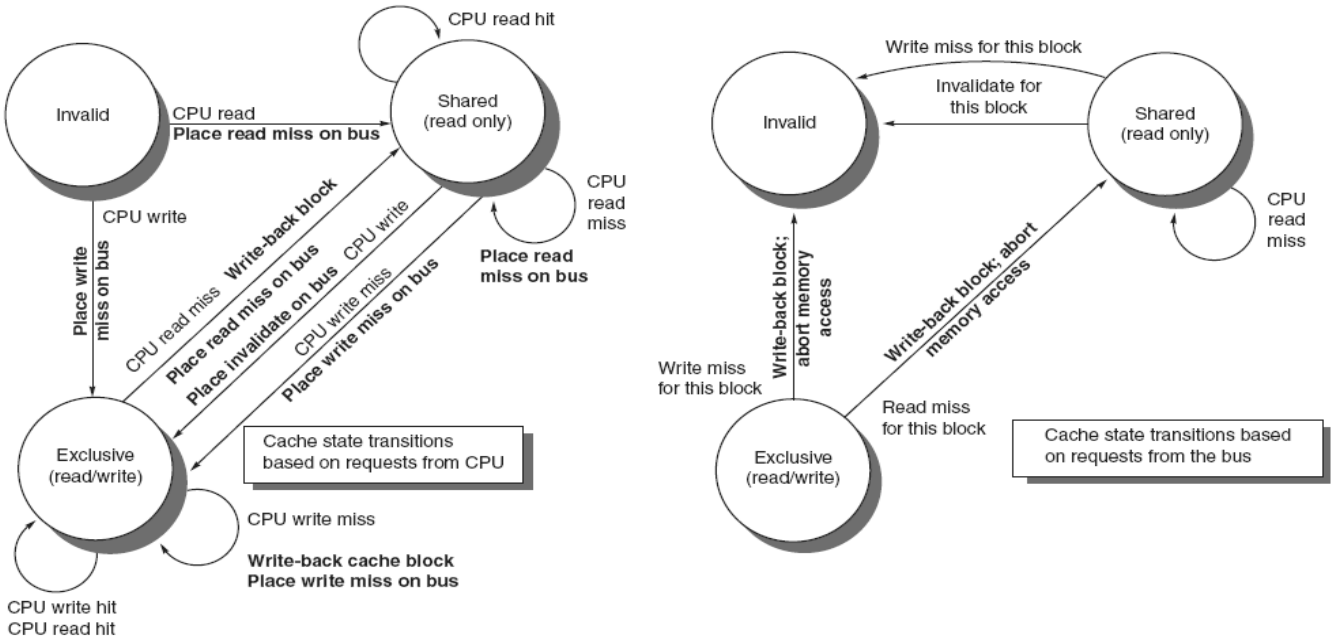
Universidad Carlos III de Madrid

Department of Computer Science and Engineering
Bachelor in Computer Science and Engineering
Computer Architecture

ARCOS

Exam

Exercise 2 [2 points]: A processor with two cores and symmetric shared memory architecture uses a bus with snooping protocol. Each core has a single level of cache memory which is private for that core. Cache coherence is kept using MSI protocol. Cache memories use direct mapping and each cache line may store 64 byte blocks. Caches have a size of 1 MB.



That processor runs the following sequence of instructions, where each thread runs in a core:

- Thread 1: lw $r0, x
- Thread 1: add $r0, $r0, 1
- Thread 2: lw $t0, z
- Thread 2: add $t0, $t0, 2
- Thread 1: sw $r0, x
- Thread 2: sw $t0, z

Cache memory is initially empty. A hit in cache memory requires four clock cycles. A miss requires 100 additional cycles for miss resolution.

You are asked:

1. Determine the final state for cache memory in each processor and for main memory, assuming that variable x is stored at address 0x00104000 and variable z is stored at address 0x00104004.
2. Determine the required time for each memory access in the previous question.
3. Determine the final state for cache memory in each processor and for main memory, assuming that variable x is stored at address 0x00104000 and variable z is stored at address 0x00106004.
4. Determine the required time for each memory access in the previous question.

SOLUTION

1) In this case, variables x and z are in the same cache line.

We will asume initial values for x and z. For example, x=100 and z=200.

Thread 1: lw $r0, x

- The corresponding cache line is in state I.
- A read miss happens.
- Cache 1: Transition I -> S.
- A read miss is placed on bus. No efecto on the other cache.

Thread 1: add $r0, $r0, 1

- No memory accesses generated.

Thread 2: lw $t0, z

- Corresponding cache line is in I state.
- Read miss.
- Cache 2: Transition I->S.
- Read miss placed on bus.
- Cache 1: S-> S

Thread 2: add $t0, $t0, 2

- No effect.

Thread 1: sw $r0, x

- Value 101 written at x.
- Write. Transition in cache 1: S->M
- Invalidation placed on bus
- Cach3 2: Transition S->I

Thread 2: sw $t0, z

- Value 202 written at z.
- Cache 2: Transition I->M.
- Write miss placed on bus.
- Cache 1: M->I
- Cache 1: block write-back performed.

Final state:

- Cach3 1: Invalid block
- Cach3 2: Exclusive block (M). Values: x=101,z=202
- Memory: Values: x=101, z=200

2)

Thread 1: lw $r0, x

- Requires 4 cycles to access cache, plus 100 cycles for read miss resolution.

Thread 2: lw $t0, z

- Requires 4 cycles to access cache, plus 100 cycles for read miss resolution.

Thread 1: sw $r0, x

- Requires 4 cycles to Access cache.

Thread 2: sw $t0, z

- Requires 4 cycles to access cache, plus 100 cycles for read miss resolution, plus 100 cycles for write-back.

3) In this case variables x and z reside in different cache lines.

Thread 1: lw $r0, x

- The corresponding cache line is in I state.
- Read miss.
- Cache 1: Block(x), Transition I -> S.
- Read miss placed on bus. No effect on the other cache.

Thread 1: add $r0, $r0, 1

- No memory access generated.

Thread 2: lw $t0, z

- The corresponding cache line is in I state.
- Read miss.
- Cache 2: Block(z), Transition I->S.
- Read miss placed on bus. No effect on the other cache.

Thread 2: add $t0, $t0, 2

- No effect.

Thread 1: sw $r0, x

- Write value 101 at x.
- Write. Block(x), Transition in 1: S->M
- Invalidation placed on bus with no effect on the other cache.

Thread 2: sw $t0, z

- Write value 202 at z.
- Cache 2: Block(z), Transition S->M.
- Invalidation placed on bus with no effect on the other cache.

Afther this operations sequence, memory is not modified as there has been no write-back.

- Cache 1, Block(x), Exclusive state (M), x=101
- Cache 2, Block (z), Exclusive state (M), z=202
- Memory: x=100, z=200

4)

Thread 1: lw $r0, x

- Requires 4 cycles for cache access, plus 100 cycles for read miss resolution.

Thread 2: lw $t0, z

- Requires 4 cycles for cache access, plus 100 cycles for read miss resolution.

Thread 1: sw $r0, x

- Requires 4 cycles for cache access.

Thread 2: sw $t0, z

- Requires 4 cycles for cache access.

Exercise 3 [2 points]:

Design in C++ a lock-free queue with the following requirements:

- The queue must be implemented as a circular buffer with size to hold 100 elements.
- The queue must allow that two threads use it at the same time when one uses the queue only for inserting (push) and the other for extraction (pop).
- No synchronization mechanism relying on the OS may be used.
- Elements inserted and extracted will be strings of characters.

The queue interface must use the following interface:

```
class circular_buffer {
public:
  circular_buffer();
  ~circular_buffer();

  bool empty() const noexcept;
  bool full() const noexcept;

  void put(const std::string & x, bool last) noexcept;
  std::pair<bool,std::string> get() noexcept;

  //...
};
```

You are asked:

1. Complete the class definition with the needed private details.
2. Implement the functions defined in the interface as well as any other auxiliary function you consider necessary.

1)

```
class circular_buffer {
public:
  circular_buffer() = default;
  ~circular_buffer() = default;

  bool empty() const noexcept;
  bool full() const noexcept;

  void put(const std::string & x, bool last) noexcept;
  std::pair<bool,std::string> get() noexcept;

private:
  int next_position(int p) const noexcept;
```

```
private:
  struct item {
    bool last;
    std::string value;
  };

  static constexpr int size_ = 100;
  item buf_[size_];
  alignas(64) std::atomic<int> next_read_ {0};
  alignas(64) std::atomic<int> next_write_ {0};
};
```

2)

```
bool circular_buffer::empty() const noexcept {
  return next_read_ == next_write_;
}

bool circular_buffer::full() const noexcept {
  const int next = next_position(next_write_.load());
  return next == next_read_.load();
}

void circular_buffer::put(const std::string & x, bool last) noexcept {
  const int next = next_position(next_write_.load());
  while (next == next_read_.load()) {
    ;
  }
  buf_[next_write_.load()] = item{last,x};
  next_write_.store(next);
}

std::pair<bool,std::string> circular_buffer::get() noexcept {
  while (empty()) {
    ;
  }
  auto res = buf_[next_read_.load()];
  next_read_.store(next_position(next_read_.load()));
  return std::make_pair(res.last,res.value);
}

int circular_buffer::next_position(int p) const noexcept {
  return p + ((p+1>=size_)?(1-size_):1);
}
```