Exercises 6
Parallel Processors and Programs

Computer Organization and Components / Datorteknik och komponenter (IS1500), 9 hp
Computer Hardware Engineering / Datorteknik, grundkurs (IS1200), 7.5 hp

KTH Royal Institute of Technology
Monday 19th February, 2018

Suggested Solutions

Concurrency, Parallelism, and Concepts

1. (a) The embedded system is using concurrent execution without parallelism. The two tasks are executed in turn. The exercise did not say anything about what happens if one of the tasks is invoked (released) while the other task is executing. One scenario is that the executing task is preempted (interrupted) by the other task and then continues after that the other task has finished. The important thing from a real-time system perspective is that both tasks can finish their execution within the period, so that a new instance of the task can be executed again. In such a case, the system is said to be *schedulable*. Note that no execution is happening in parallel, the system switches execution contexts between the two tasks.

   (b) The numerical program is executing sequentially on a processor, but performs several operations in parallel. Note that the exercise did not say that the program is using more than one thread, so parallelism is achieved using data-level parallelism, for instance by using SIMD instructions.

   (c) The web server is executing both concurrently and in parallel (if the operating system can exploit the concurrency). The concurrency is made explicit by the server by creating new threads when HTTP requests arrive. The OS may be able to execute the potentially independent tasks on separate cores.

2. (a) SISD = Single Instruction Stream, Single Data Stream. SIMD = Single Instruction Stream, Multiple Data Stream, MISD = Multiple Instruction Stream, Single Data Stream, and MIMD = Multiple Instruction Stream, Multiple Data Stream.

   (b)
   - MIPS Uniprocessor: Should be placed in SISD. This processor reads only one stream of instructions and does not manipulate multiple data items.
   - Task-level parallelism: Could be placed inside MIMD. Several different tasks (with separate instruction streams) are executed in parallel. Each such task can manipulate at least one data stream each.
   - AVX: Advanced Vector Extension (AVX) is a SIMD extension by Intel that was released in 2011. A single operation can handle eight 32-bit floating-point operations or four 64-bit floating-point operations.
   - Intel Core i7: An Intel Core i7 has several cores and is clearly MIMD. Note that each of the cores also has AVX instructions, so each core can be seen as SIMD.

- GPU: A Graphical Processing Unit (GPU) has all kinds of parallelism, including both SIMD and MIMD.
- Data-level parallelism: Belongs to SIMD, that is, we have one stream of instructions, but can perform that instruction on multiple streams of data.
- ILP: Instruction Level Parallelism (ILP) is basically orthogonal to this matrix, and may be placed in all the four squares in the matrix. It improves the execution so that several instructions can be executed in parallel. ILP can be used both in the case when there is just one stream of instruction, or in the case of multiple streams.

Speedup and Amdahl's law

3. It is important to compare with the other efficient sequential implementation I_s, but this comparison must be done on exactly the same hardware. Hence, all benchmarking should be done on the 8 core machine. The following steps describe one potential way of doing the evaluation.

   (a) Disable hardware multi-threading (can be done on most modern operating systems).

   (b) Disable all cores but one. Run I_s and save the measurement as m_s.

   (c) In turn, run I_p with $1, 2, \ldots, 8$ cores enabled, and save the values as $m_{p1}, \ldots, m_{p8}$.

   (d) Plot a speedup graph (speedup on the Y-axis and the number of cores on the X-axis), that is, the plot should have points at values $\frac{m_s}{m_{p1}}, \ldots, \frac{m_s}{m_{p8}}$. Note that it is important to use m_s for the numerator to get true speedup comparison. If m_{p1} was used as the numerator, the plot would show relative speedup.

   (e) Enable hardware multi-threading, and perform the same measurement and plotting as above for 16 cores. Note that the OS believes that there are 16 cores, but 8 of them is just an effect of hardware multi-threading. Nevertheless, multi-threading can give performance gains since it can hide latencies due to, e.g., cache misses.

   If it is not possible to disable cores in the OS (or in the BIOS), you may test your application by just running your parallel program using 1 to 16 software threads (enabling one software thread in each measurement). However, it is better to enable and disable cores, if possible.

4. (a) Apply Amdahl's law and plot the graph using function

$$S(N) = \frac{1}{\frac{1-0.1}{N \times 0.8} + 0.1} \tag{1}$$

where $S(N)$ is the speedup, as a result of using N cores. You should plot the graph for $N > 1$. For one core, we do not get any speedup. Hence $S(1) = 1$.

(b) 10

(c) It is called *weak scaling* when the program size increases proportionally to the number of processors. The example in (a) is using *strong scaling* where the problem size is fixed. An argument for weak scaling is that larger problems need more data. If we can handle bigger problems with more cores, we can solve more interesting problems. An argument for strong scaling is that for a given problem size, there is a limit on how much speedup we can achieve, regardless of the number of cores we add. Both scaling approaches make sense in their own contexts.

Instruction Level Parallelism

5. (a) The function takes an array of integers (parameter `src`) of length `n`, reverses this array, multiples each element with 2, and then writes this back to parameter `dst`.

(b) One potential translation is:

```
reverse_double:
        addi $t0, $0, 0       # Count i is $t0. Initiated to 0
loop:
        slt  $t1, $t0, $a2    # Check if i < n
        beq  $t1, $0, done    # If not true, branch to done

        sub  $t1, $a2, $t0    # Compute n - i
        sll  $t1, $t1, 2      # Multiply with 4
        add  $t1, $t1, $a0    # Add the base address
        lw   $t2, -4($t1)     # Load word with address
                              # From n - i - 1

        sll  $t2, $t2, 1      # Multiply the value with 2

        sll  $t1, $t0, 2      # Multiply i with 4 to get the address
        add  $t1, $a1, $t1    # Destination address
        sw   $t2, 0($t1)      # Store the word, dst[i]

        addi $t0, $t0, 1      # Increment i with one
        j    loop             # Run the for loop again
done:
        jr   $ra              # Function return
```

Some comments:

- The arguments of the function are stored in the standard argument registers: `$a0` = first argument corresponding to C parameter `src`, `$a1` = second argument corresponding to C parameter `dst`, and `$a2` = third argument corresponding to C parameter `n`.

- Note that we have used the temporary registers `$t1`, `$t2`, and `$t3`, and not the save registers (for instance `$s1`) because they do not need to be saved on the stack by the callee.
- Note that if we switch the order of the two last `sll` instructions, we remove a dependency between the `lw` instruction and the first `sll` instruction. However, we show the unoptimized version here so that the next answer shows how a data hazard can be handled using stalling.

(c) The inner loop takes 15 cycles for executing 12 instructions. Note that the load instruction needs to stall one cycle because the result is needed in the next cycle. Note also that the jump instruction takes 2 cycles. Moreover, the `beq` instruction needs to stall one cycle because of its direct dependency to the result of the `slt` instruction (the reason is that the check for equality is done in the decode stage). Before the loop, there is one instruction that takes 1 cycle. The loop will execute 5 times. The 6th time the loop starts, `slt` will return 0 in `$t1`. As a consequence, the program counter will jump to `done`. The last two instructions (`slt` and `beq`) take 4 cycles because the dependency between the two instructions (one stall) and the branch is taken (stall because of control hazard). Finally, the function return instruction takes 2 cycles. In summary, we execute $1 + 12 \times 5 + 2 + 1 = 64$ instructions. The number of cycles are $1 + 15 * 5 + 4 + 2 = 82$. Hence, the IPC is $\frac{64}{82} \approx 0.78$

(d) The updated 2-issue code is as follows:

```
    SLOT 1                  SLOT 2                CLOCK CYCLE

reverse_double:
    addi $t0, $0, 0                                   # 1
loop:
    sub  $t1, $a2, $t0      slt  $t2, $t0, $a2        # 2

    sll  $t1, $t1, 2        beq  $t2, $0, done        # 4

    add  $t1, $t1, $a0                                # 5

    lw   $t2, -4($t1)                                 # 6

    sll  $t2, $t2, 1        sll  $t1, $t0, 2          # 8

    add  $t1, $a1, $t1      addi $t0, $t0, 1          # 9

    sw   $t2, 0($t1)        j    loop                 # 10
done:
    jr   $ra
```

Some comments:

- When putting `sub` and `sll` in parallel with `slt` and `beq`, respectively, we had to rename the target register for `slt` to `$t2`.
- Note that the `lw` instruction will stall since the result `$t2` is used directly after the instruction.

- The `beq` instruction needs to stall one clock cycle because the comparison is done in the decode stage and there is a direct data dependency to the `slt` instruction.

(e) The inner loop takes 10 cycles to execute 12 instructions. There is one instruction executed before the loop and the function return instruction takes 2 cycles. The last (6th) time the loop is entered, 4 instructions will be executed and 4 cycles consumed. Note, however, that only two of these instructions are actually used, so for fairness, we only include 2 of the instructions (`slt` and `beq`) in the calculation.

In summary, we execute $1 + 12 \times 5 + 2 + 1 = 64$ instructions. The number of cycles are $1 + 10 * 5 + 4 + 2 = 57$. Hence, the IPC is $\frac{64}{57} \approx 1.22$

(f) Speedup is defined as the execution time it takes before the improvement, divided by the execution time after the improvement. In this case, we can use the number of clock cycles as the measurement of time, presupposed that the two examples use the same clock frequency. From exercise 5c we know that the program took 82 clock cycles using an 1-issue pipeline, whereas we have 52 clock cycles for an 2-issue processor (see exercise 5e). Hence, the speedup is $82/52 \approx 1.58$

Concurrent Programming and Semaphores

6. Assume that there is a FIFO buffer `buffer` (also called a FIFO queue). There are two possible operations on the buffer, `enqueue(b,e)` that adds a data element `e` to the back of the FIFO buffer `b`, and `dequeue(b)` that removes the front element from a FIFO buffer `b` and returns the removed element. Assume that the queue can hold between 0 and `n` elements. The behavior is undefined if either `enqueue(b,e)` is called when the buffer is full (there are `n` number of elements in `b`) or if `dequeue(b)` is called when the buffer is empty.

   The problem is twofold: (i) We need to make sure that the calls to `enqueue` and `dequeue` are atomic, and (ii) the undefined behaviors of removing from an empty buffer or adding to a full buffer must not happen.

   To solve this problem, we define three semaphores. We define one binary semaphore called `mutex` that is initiated to value 1. This semaphore is used to solve problem (i). We then define two more semaphores for counting elements in the queue. Semaphore `available_space` keeps track of the number elements that can be stored in the FIFO buffer. We also define a semaphore `number_of_items` that keeps track of how many items that are stored in the buffer. Semaphore `mutex` is initiated to 1, meaning that it is available from the beginning. Semaphores `available_space` and `number_of_items` are initiated to `n` and 0, respectively.

   We use the classic two functions `P(s)` that waits or decrements the semaphore, or `V(s)` that increments the semaphore[1].

   The producer function can now be defined as follows:

[1] Why are the functions called P and V? Semaphores were invented by Edsger Dijkstra who was originally from the Netherlands. In Dutch *Proberen* means "to test" and *Verhogen* means "to increment".

```
function insertItem(buffer, item) returns nothing{
   P(available_space)     //Waits until there is available space
   P(mutex)               //Locks the buffer
   enqueue(buffer, item)  //Adds the item to the FIFO buffer
   V(mutex)               //Releases the lock
   V(number_of_items)     //Tells the consumer thread that there is
                          //one more item
}
```

Note that `P(available_space)` waits until `available_space` becomes larger than 0 and when it does, decrements it. The `mutex` makes sure that only one thread is accessing the buffer at the same time. Finally, `V(number_of_items)` increments `number_of_items` by one, meaning that there is one more item in the buffer.

The consumer function is defined as follows:

```
function removeItem(buffer) returns item{
   P(number_of_items)     //Waits until there is an available item
   P(mutex)               //Locks the buffer
   item = dequeue(buffer) //Adds the item to the FIFO buffer
   V(mutex)               //Releases the lock
   V(available_space)     //Tells the consumer thread that there is
                          //now more space available
   return item
}
```

The `removeItem` function is very similar to the previous function. Line `P(number_of_items)` waits until there is an item available, i.e., that `number_of_items` is larger than zero. When it does, it decrements the semaphore. After (safely) dequeuing an item from the buffer, `V(available_space)` increments semaphore `available_space`.

Data-Level Parallelism

7. (a) This is an example of Intel's Advanced Vector Extension (AVX) assembly code.

   (b) As usual, each line represents one assembly instruction. In the following, we describe each line of code:

   ```
   vmovapd  (%r10), %ymm0
   ```

   Moves 256 bits from the memory address that register `%r10` points to into the SIMD register `%ymm0`. The `pd` means *packed double*, that is, it moves four double precision floating-point values (each double is 64-bit).

   ```
   vmovapd  (%r11), %ymm1
   ```

   Same as the first line, but reads from the address in `%r11` and writes to `%ymm1`.

   ```
   vmulpd   %ymm0,  %ymm1, %ymm1
   ```

   Performed four double precision floating-point multiplications in parallel. Note that the two first operands are the source operands and that the destination is to right. That is, the function multiplies `%ymm0` with `%ymm1` and stores the result in `%ymm1`.

   ```
   vaddpd   (%r10), %ymm1, %ymm1
   ```

   Adds four doubles located at the address in `%r10` with `%ymm1` and stores the result in `%ymm1`.

   ```
   vmovapd  %ymm1, (%r11)
   ```

   Stores the value in `%ymm1` into memory at the address in `%r11`.