ASIP Laboratory - Session 3

Paul Georg Wagner Majd Mansour

May 24, 2017

Exercise 1: BubbleSort_Index

a) *How often the code of the inner loop is executed (not only the exchange part, the whole inner loop)? Please do not only answer this question, but go through the output step by step. First you should look at the code and think about the answer and then you should add a counter to the code to compute the correct result just to make sure, your prediction is correct.*

The given bubble sort implementation uses two nested for-loops to calculate the sorted array. The outer loop (line 22) divides the unsorted area at the beginning of the array from the sorted area at the end. The outer loop index `i` specifies the location of the sorted area, counted from the end of the considered array. With each iteration of the outer loop, the maximum value of the unsorted area will be shifted to the beginning of the growing sorted area.

The inner loop (line 27) iterates the unsorted area of the array and swaps two adjacent elements if they are not in order. This causes higher array elements ("bubbles") to move towards the end of the array.

In order to calculate the number of inner loop iterations, we have to consider how an array of length n is processed by this algorithm. When the outer loop is executed the very first time, there are no already sorted elements, so each of the n array elements belongs to the unsorted area. Hence there are $n-1$ many adjacent pairs that the inner loop will iterate over. So the first outer loop execution results in $n-1$ inner loop executions.

After the first outer loop execution has been finished, the biggest element will be placed as the last element in the sorted area, so there are only $n-1$ many unsorted elements left. This means that the next outer loop execution results in only $n-2$ many inner loop executions, since there are only that many adjacent pairs left in the unsorted area. The final outer loop execution then results to only one inner loop iteration (i.e. only the first adjacent pair is left to compare).

So all in all, the total amount of inner loop iterations results to

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$$

With an array size of 20 elements – as is given in the provided example – we get a total of $\frac{20*19}{2}=190$ inner loop iterations. This is exactly the result that we get if we insert a counter after line 27 in order to record the number of inner loop iterations.

Exercise 2: BubbleSort_Address

a) *How many load- and how many store- instructions are executed for each inner loop (distinguish between when there is exchange and no exchange)? Compare the index-*

version against the address-version and mention the two main points, why the address-version needs less memory accesses.

The algorithm given in *BubbleSort_ Index.c* executes two read accesses on the array in the inner loop (line 28), another two read accesses in the if-clause of the inner loop (lines 32 and 33) and two write accesses also in the if-clause of the inner loop (lines 33 and 34). The array accesses in the if-clause are conditional in the sense that they are only executed if the two currently compared array elements are not in the right order. Hence the number of array accesses differ from the best case (array is already sorted) to the worst case (array is reversely sorted).

In the best case the condition of the if-clause in line 28 is never true, since all the elements are already in order. Hence the read and write accesses inside the if-clause are never executed. Nevertheless, the two read accesses in the condition are still performed. Since the inner loop is executed $\frac{n(n-1)}{2}$ times, we end up with a total of $2 * \frac{n(n-1)}{2} = n(n-1)$ many read accesses and no write accesses in the best case.

In the worst case the condition of the if-clause always holds true, since each two adjacent array elements are reversely ordered. Hence we end up with four read accesses and two write accesses in each inner loop execution instead of just two read accesses like in the best case. This means that in the worst case we end up with a total of $4 * \frac{n(n-1)}{2} = 2 * n(n-1)$ many read accesses and $2 * \frac{n(n-1)}{2} = n(n-1)$ many write accesses.

The algorithm given in *BubbleSort_ Address.c* makes some optimizations in the way the algorithm is implemented in order to save some of those memory accesses. This version executes one read access in the outer loop (line 32), one read access in the inner loop (line 37) as well as two write access in the if-clause of the inner loop (lines 42 and 43). This modified version of the algorithm saves memory accesses by not reading the same array value twice in one loop iteration, as the previous version does. It furthermore moves one of the read accesses from the inner to the outer loop, resulting in reduced memory accesses. This can be done because the optimized algorithm keeps the values of the compared array elements in local variables and shifts them for each inner loop iteration (lines 48 and 49). That way only one read access on the array is required in the inner loop.

In the best case only the read accesses are executed, as the if-condition that protects the two write accesses never holds true. Since the outer loop is executed n times, we get a total of $n + \frac{n(n-1)}{2}$ many read accesses and no write access for the best case. In the worst case also the two write accesses are executed at each iteration of the inner loop. This means that $2 * \frac{n(n-1)}{2} = n(n-1)$ many write accesses are executed while the number of read accesses remains unchanged.

The number of read and write accesses in the various cases are presented in table 1. All in all it is apparent that the optimized version in *BubbleSort_ Address.c* in general takes less read operations while the amount of write operations is the same for both algorithms.

	BubbleSort_ Index.c		*BubbleSort_ Address.c*	
	Best case	Worst case	Best case	Worst case
read	$n(n-1)$	$2 * n(n-1)$	$n + \frac{n(n-1)}{2}$	$n + \frac{n(n-1)}{2}$
write	0	$n(n-1)$	0	$n(n-1)$

Table 1: Complexity of bubble sort variants.

Exercise 3: Translation

a) *How many cycles do you need for execution in dlxsim? Attach "bs_ NoPipeline.s" to the mail.*

In order to translate C code to assembly, several points have to be considered. The C code uses two nested while-loops, which are translated using branch and jump instructions. The outer loop head is defined in lines 4 and 5, where a `beqz` instruction is used to jump out of the loop if the loop condition does not hold anymore. At the end of the loop body an unconditional jump directs the control flow back to the loop head, completing the loop (line 30). Similarly, the inner loop is structured (lines 11, 12 and 26).

Furthermore, there is no instruction available to directly move the contents of a register to another register. This behavior is implemented using the `addu` instruction with one of the parameters as the zero register.

Finally, the behavior of pointer increment operations in C have to be studied carefully in order to translate them to assembly. When a pointer is incremented in C, the value of the pointer changes according to the type of data it points to. For example, if a pointer to a data structure with a size of four bytes is incremented, the value of the pointer is increased by four instead of one, in order to point to the beginning of the next data structure. This behavior has to be considered when translating instructions like `next_j++` to assembly. Since `next_j` is a pointer to a four byte integer, the corresponding assembly instruction is `addui %r7, %r7, $4` (line 14). Similarly, for translating the instruction `innerLoopEnd = array + endIndex` the value in `endIndex` has to be multiplied by four before adding it to `array`. In our implementation, this is done by shifting the respective value two digits to the left, using the `slli` instruction (e.g. line 1 and 7).

The result of the executed translation is given in listing 1 and also in the file *bs_ NoPipeline.s*. This implementation takes a total of 3830 instruction cycles to finish.

Listing 1: bs_NoPipeline.s.

```
1                       slli %r1, %r3, $2
2                       addu %r6, %r10, %r1
3                       addu %r4, %r0, %r2
4   outer_loop_begin:   sltu %r1, %r4, %r3
5                       beqz %r1, outer_loop_end
6                       nop
7                       slli %r1, %r2, $2
8                       addu %r5, %r10, %r1
9                       lw %r8, 0(%r5)
10                      addu %r7, %r0, %r5
11  inner_loop_begin:   sltu %r1, %r5, %r6
12                      beqz %r1, inner_loop_end
13                      nop
14                      addui %r7, %r7, $4
15                      lw %r9, 0(%r7)
16                      sltu %r1, %r9, %r8
17                      beqz %r1, if_end
18                      nop
19                      sw 0(%r5), %r9
20                      sw 0(%r7), %r8
21                      addu %r1, %r0, %r8
```

```
22                              addu %r8, %r0, %r9
23                              addu %r9, %r0, %r1
24          if_end:             addu %r5, %r0, %r7
25                              addu %r8, %r0, %r9
26                              j inner_loop_begin
27                              nop
28          inner_loop_end:     subui %r6, %r6, $4
29                              addui %r4, %r4, $1
30                              j outer_loop_begin
31                              nop
32          outer_loop_end:
```

Exercise 4: Adding the pipeline model

a) *How many cycles do you need for execution in dlxsim and ModelSim? Attach "bs_basis.s" to the mail.*

Unlike in the previous exercise, in order to execute the code on the provided CPU, the pipeline delay between the register fetch and the register write-back of an instruction has to be considered. While in bs_NoPipeline.s it is assumed that the new value of a written register is available for reading in the very next cycle, this is not the case on the real CPU. Due to the pipeline structure of the CPU, a delay of three cycles exists between the register fetch and the write-back phase of an instruction. This means that between each true data dependency (read after write) at least three instructions have to be placed in the code in order to give the writing instruction enough time to write back the correct result, before it is read again. This delay can be achieved by either putting three `nop` instructions or any other meaningful instruction without dependencies between the dependent instructions.

In order to get a version of the assembly code that resolves the pipeline conflicts occurring in the original version, three `nop` instructions are placed between each data dependency. Sometimes also two `nop` instructions are sufficient, if there already is an independent instruction between them (e.g. between `addu %r1, %r0, %r8` and `addu %r9, %r0, %r1`). Furthermore, data dependencies across jumps must also be considered. Even if a jump could alter the instruction flow, the rule that three other instructions need to reside between the dependent instructions must still be fulfilled. This leads to a single `nop` being placed right in front of the `j outer_loop_begin` instruction, in order to put three instructions (`nop`, `j outer_loop_begin` and another `nop`) between the dependent instructions `addui %r4, %r4, $1` and `sltu %r1, %r4, %r3`. In this case only one `nop` has to be added, since the `nop` after the jump instruction is also executed due to the pipeline design.

The resulting code that resolves the pipeline conflicts using a minimal amount of `nop` instructions is given in the file *bs_basis.s*. As the amount of instructions increased, the modified code now needs a total of 6941 cycles to finish, measured with *dlxsim*. In order to proof that the modified code also runs on the actual CPU, we also simulated it using *ModelSim* (see figure 1). With *ModelSim* the code required a total of 6943 cycles. The discrepancy of two cycles most likely is caused by additional instructions that the *ModelSim* compiler introduces for the simulation.

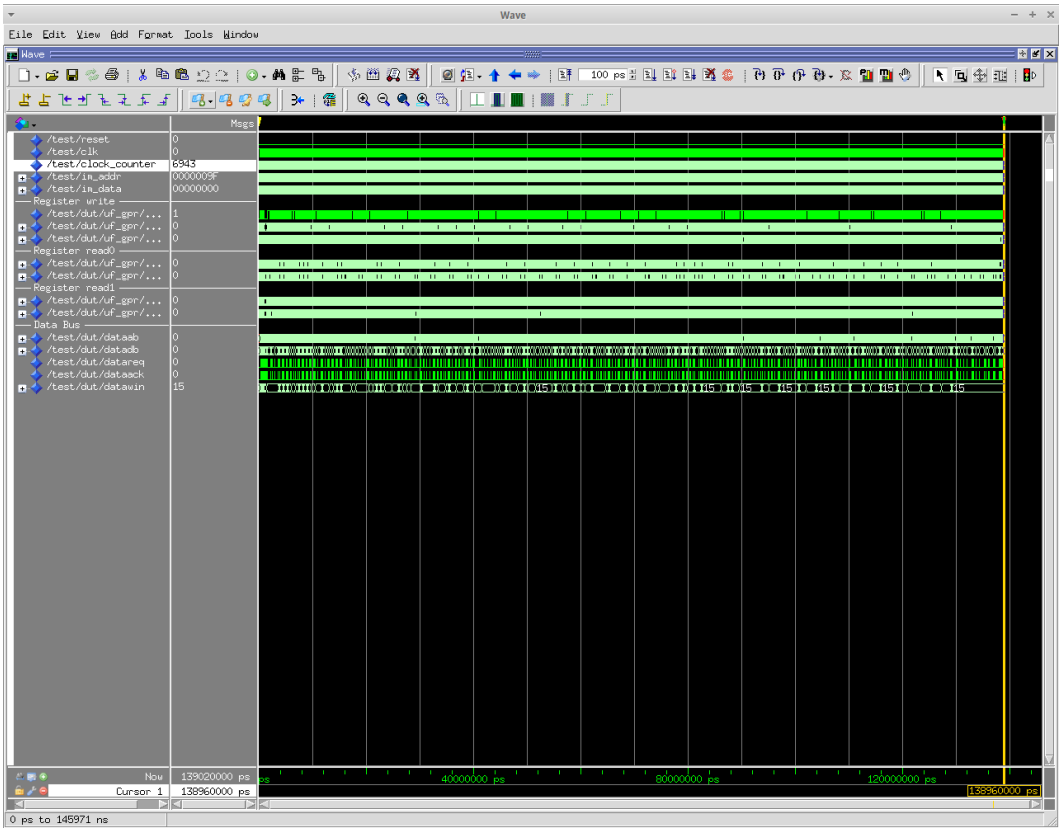

Figure 1: *bs_ basis.s* executed using *ModelSim*.