Computer Architecture and Programming Languages
Jacobs University Bremen
Romelda Blaceri

Course: CO20-320241
2 December, 2019

Homework 11

Problem 11.1

Solution:
Considering the table from the slides, we have:

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load Word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store Word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

a) Multi-cycle implementations allow functional units to be used more than once per instruction as long as they are used on different clock cycles. The clock cycle is used efficiently in this case: it is timed to accommodate the slowest instruction step (200 ps in our case). The instructions we have take the following number of cycles:

* `lw` $\rightarrow$ 5 cycles (since all the cells in that row of the above table are filled).
* `sw` $\rightarrow$ 4 cycles.
* The R-format instructions $\rightarrow$ 4 cycles.
* The branching instruction $\rightarrow$ 3 cycles.

We calculate the total by multiplying the number of cycles by the number of time that instruction is performed (2 load instructions, 1 store instruction, 3 R-format instructions and 1 branching instruction), and then we multiply the whole sum by 200 ps. Therefore, the time taken for the process using multi-cycle implementation is:

$T = (2 \cdot n_{lw} + 1 \cdot n_{sw} + 3 \cdot n_{\text{R-format}} + 1 \cdot n_{\text{Branch}}) \cdot 200 \text{ ps} = (2 \cdot 5 + 1 \cdot 4 + 3 \cdot 4 + 1 \cdot 3) \cdot 200 = 5800 \text{ ps}$

With the single cycle approach, every instruction must be executed within one clock cycle, where the time of this clock cycle is dependent on the time of the slowest instruction. In our case, the slowest instruction is `lw` (800 ps). Therefore, every instruction will take 800 ps to execute. We have in total 7 instructions so, the total time it takes to execute the program using a single cycle approach is: $7 \cdot 800 = 5600$ ps.

In our case, the single cycle approach is faster than the multi-cycle. Performing the calculations ($5800/5600 - 1 \approx 0.0357$), we find that it is approximately 3.57% faster.

b) For the single pipelined approach, each instruction path starts right after the previous Instruction Fetch (which takes 200 ps) is finished. In our case, we have 7 instructions, and the largest one takes 5 cycles, so in total we have 7 times Instruction Fetch, plus 5 times 200 ps for the longest path, minus 200 ps, since in the longest path we're counting the Instruction Fetch again. Therefore, the total time is: $T = (7+5-1) \cdot 200 \text{ ps} = 2200 \text{ ps}$
We have already calculated the single cycle approach, and it is clear that for this case the pipeline approach is faster (performing the calculations, it's 154.54% faster).

c) For the multi-cycle pipelined approach, we need to have an order in which the instructions will be performed. Since no such order is explicitly given, we consider the order in which the instructions are written (load, load, store, 3 R-format instructions and branching instruction). Again, each instruction path starts right after the previous Instruction Fetch (which takes 200 ps) is performed. The last instruction is the branch instruction, which has 3 cycles. Therefore, in total we have 6 times Instruction Fetch plus 3 times 200 ps for branch (we consider all the cycle times as 200 ps since that is the slowest cycle time). So, we have: $T = 9 \cdot 200 \text{ ps} = 1800 \text{ ps}$
We have calculated both the approaches, and again, pipeline approach is much faster then the multi-cycle approach (222.22% faster).

Problem 11.2

Solution:

a) In order to recognize string `$zero`, we write: **"$zero"**

b) To recognize any character which is either a letter or a digit, we use [a-zA-Z0-9]. If we want to say that either one of those characters is included, or none of them, we add * at the end, and since we want the string to start with a and end with b, the final expression is: **a[a-zA-Z0-9]*b**

c) Since the string should start and end with a digit, we have [0-9] in the beginning and in the end. Since it may contain letters, digits, and underscores (so it may also not contain any of them), we use [a-zA-Z0-9_]*, and the final expression would be: **[0-9][a-zA-Z0-9_]*[0-9]**

d) The beginning of the string should be abb. Then we have at least 4 a's, so aaaa. Then we may have a or b, so [ab], such that we don;t exceed 10 characters overall (we already have 7 characters with abb and the 4 a's, so the remaining 3 can be either a or b → notation 0,3). As a result, the expression is: **abbaaaa[ab]{0,3}**

e) If we want to include all positive integer numbers, we may have 2 cases:
* if 0 is not included, we can write the possibilities of digits: [1-9] in the beginning, since we cannot allow something like 023 for example, and then add [0-9]*, to show that we can have as many digits as we want to express a specific number: **[1-9][0-9]***.
* if 0 is included, we write the above expression, or zero: **([1-9][0-9]*|0)**

f) To include all integer numbers we have to include the possible minus sign in the beginning (-?), and the expression then will follow same as above: **(-?[1-9][0-9]*|0)**

g) For the floating point numbers, we have three cases:
* Case when the number is bigger or equal to 1 (so that we avoid something like 002.5), we write: [1-9][0-9]*(.[0-9]*)? where we include cases: whole number (e.g. 12, 35), numbers having just the point to express that the floating part is 0 (e.g. 5. or 78.), and normal floating point number bigger than 1 (e.g. 45.678).
* Case when we have normal floating point numbers smaller than 1 (e.g. 0.23, 0.0156): 0(.[0-9]*)?
* Case when we have just the floating part, indicating that the first part is zero (e.g. .5 or .078) for numbers that are smaller than one: .[0-9]+
Therefore, the final expression would be: **([1-9][0-9]*(\.[0-9]*)?|0(\.[0-9]*)?|\.[0-9]+)**
Note that \. is written to use the point character (.)

h) Since all of the strings that it should not recognize start with p and just one of the strings that it should recognize starts also with p, we just write that specific string as a unique case: "pit". The other strings start with r, s or t, so we have [rst]+ so that we're sure that the word will have the same start and not p, and then the remaining part can be [a-z]*, which satisfies the other part of these strings (the space is needed for the string: slap two). So, the final expression will be: **("pit"|[rst]+[a-z]*)**

Problem 11.3

Solution:

a) The string should start with a and then be followed by at least one b and end with c, so **(1)** is the correct answer (abc). therefore the correct answer is (1).

b) The string must start with a, followed by any character but newline, then followed by at least one b or c. Then at least one of b or c characters follow. Therefore, the ones that match are all (**(1), (2), (3), (4), (6)**) except from (5) since the string must have at least 3 characters (a, any character, b or c).

c) The string must start with "very", followed by a space at least one time, it may be then followed by "happy" and a space, then it should contain one of the following: CS, IMS, or ECE and in the end it should have "student". So, the strings that satisfy this are: **(3)** (very very happy ECE student), **(4)** (very very very happy IMS student) and **(5)** (very very very very IMS student).

d) The string has to start with a <and end with a >. In the middle it should contain at least one character except from >, so the ones that satisfy the condition are: **(1)** (<an xml tag>), **(3)** (</closetag>) and **(5)** (<with attribute="77">).