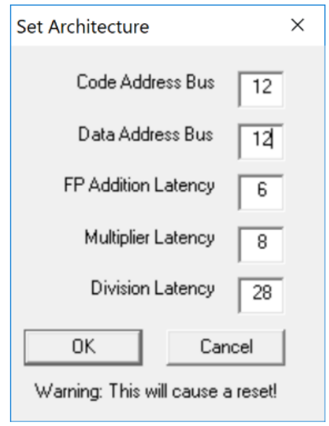
Architetture dei Sistemi di Elaborazione [AAA-GRA]	Delivery date: October 25th 2022
Laboratory 2	Expected delivery of lab_02.zip must include: - **`program_1.s`** and **`program_2.s`** - This file, filled with information and possibly compiled in a pdf format.

Please, configure the winMIPS64 simulator with the *Base Configuration* provided in the following:



- Code address bus: 12
- Data address bus: 12
- Pipelined FP arithmetic unit (latency): 6 stages
- Pipelined multiplier unit (latency): 8 stages
- divider unit (latency): not pipelined unit, 24 clock cycles
- Forwarding is enabled
- Branch prediction is disabled
- Branch delay slot is disabled
- *Integer ALU: 1 clock cycle*
- *Data memory: 1 clock cycle*
- *Branch delay slot: 1 clock cycle.*

1) Write an assembly program (**`program_1.s`**) for the *winMIPS64* architecture described before able to implement the following piece of code described at high-level:

```
for (i = 0; i < 60; i++){
        v5[i] = ((v1[i]+v2[i]) * v3[i])+v4[i];
        v6[i] = v5[i]/(v4[i]*v1[i]);
        v7[i] = v6[i]*(v2[i]+v3[i]);
}
```

Assume that the vectors v1[], v2[], v3[], and v4[] are allocated previously in memory and contain 60 double precision **floating point** values; assume also that (v4[i]*v1[i]) does not contain 0 values. Additionally, the vectors v5[], v6[], v7[] are empty vectors also allocated in memory.

a. Using the simulator and the *Base Configuration*, disable the Forwarding option and compute how many clock cycles the program takes to execute.

	Number of clock cycles
`program_1.S`	5685

Enable one at a time the **optimization features** that were initially disabled and collect statistics to fill the following table (fill all required data in the table before exporting this file to pdf format to be delivered).

Table 1: **Program performance for different processor configurations**

	Number of clock cycles

Program	Forwarding	Branch Target Buffer	Delay Slot	Forwarding + Branch Target Buffer
`program_1`	4087	5631	117	4033

2) Write an assembly program (**`program_2.s`**) for the winMIPS64 architecture able to compute the 2D Convolution between 5x5 and 3x3 matrices, and store the result in a 3x3 matrix**.** The 2D convolution is a frequently used operation in many fields, such as image processing or deep learning algorithms.
The inputs of the program are the following: a single 5x5 image, also known as Input Feature Map (ifmap) in deep learning models, and a 3x3 filter (also known as kernel). In a 2D Convolution, this kernel slides over the 2D input image and performs an elementwise multiplication with the part of the input it is currently on, and then it sums up the results into a single output pixel. This operation between the ifmap and the kernel produces a 3x3 output matrix, also known as output feature map (ofmap).

As an example (Figure 1), to compute the first element in the output matrix (y), the following operations must be performed:

$$y[0][0] = x[0][0]*h[0][0] + x[0][1]*h[0][1] + x[0][2]*h[0][2] + x[1][0]*h[1][0]+ x[1][1]*h[1][1]+ x[1][2]*h[1][2]+ x[2][0]*h[2][0]+ x[2][1]*h[2][1]+ x[2][2]*h[2][2] = 24$$

A graphic example is illustrated below.

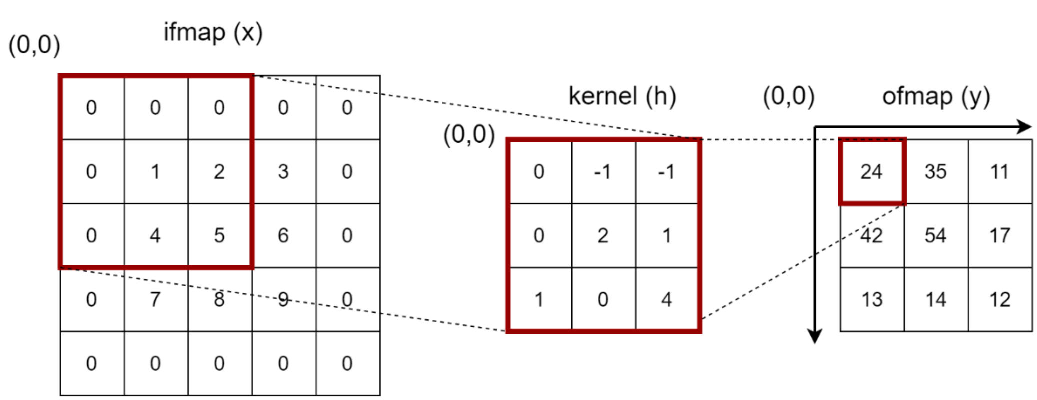


Figure 1: 2D Convolution

As shown in Figure 2, to compute the second element of the ofmap, the kernel must slide over the ifmap and the following operations must be performed:

$$y[0][1] = x[0][1]*h[0][0] + x[0][2]*h[0][1] + x[0][3]*h[0][2] + x[1][1]*h[1][2]+ x[1][2]*h[1][1]+ x[1][3]*h[1][2]+ x[2][1]*h[2][0]+ x[2][2]*h[2][1]+ x[2][3]*h[2][2] = 35$$

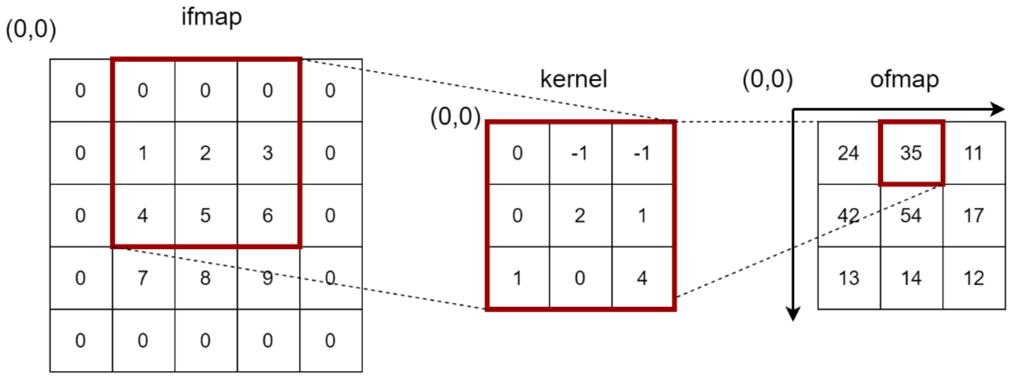
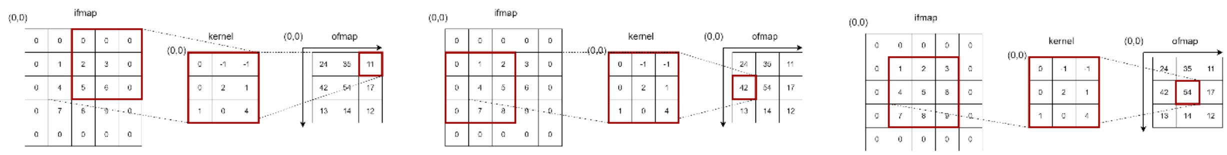


Figure 2: 2D Convolution - second element of the ofmap

This operation is performed for every element of the ofmap (as shown below for some extra pixels).



Note that the kernel should not be inverted, suppose that it has the correct shape to convolve over the input image.

Write an assembly program that accepts two matrices and returns one 3x3 matrix holding the result of the 2D convolution. Each element of the input matrices is a 8-bit signed integer. The ofmap should include elements in the same data representation.

<u>MIPS64 Example</u> (you can initialize matrices as desired):

ifmap: .byte 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
kernel: .byte 1,1,1,1,1,1,1,1,1
ofmap: .space 9

The ofmap, in this case, will be equal to:
ofmap: .byte 9,9,9,9,9,9,9,9,9

3) Using the WinMIPS64 simulator, validate experimentally the Amdahl's law, defined as follows:

$$\text{speedup}_{\text{overall}} = \frac{\text{execution time}_{\text{old}}}{\text{execution time}_{\text{new}}} = \frac{1}{(1-\text{fraction}_{\text{enhanced}}) + \dfrac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}$$

a. Using the program developed before: **`program_1.s`**
b. Modify the processor architectural parameters related with multicycle instructions (`Menu`→`Configure`→`Architecture`) in the following way:

1) Configuration 1
    - Starting from the *Base Configuration*, change only the FP arithmetic unit latency to 2
2) Configuration 2
    - Starting from the *Base Configuration*, change only the Multiplier unit latency to 6.
3) Configuration 3
    - Starting from the *Base Configuration*, change only the division unit latency to 10 (Multiplier unit and FP unit latency left to original value, i.e., 4 and 6, respectively)

Compute by hand (using the Amdahl's Law) and using the simulator the speed-up for any one of the previous processor configurations. Compare the obtained results and complete the following table.

Table 1: **`program_1.s` speed-up computed by hand and by simulation**

Proc. Config. / Speed-up comp.	Base config. [c.c.]	Config. 1	Config. 2	Config. 3
`By hand`	4156	1.080475885	1.09780083	1.351105332
`By simulation`	4087	1.061282783	1.09541677	1.351074380