Architetture dei Sistemi di Elaborazione 02GOLOV	Computer Architectures 02LSEYG
Laboratory 0x01	**Expected delivery of lab_01.zip including:** **- program_3.s** **- lab_01.pdf (fill and export this file to pdf)**
Delivery date 16/10/2025	

General procedure for simulating a program with gem5 and visualize the pipeline behaviour

1. If you are working on a LABINF PC, remember to boot Ubuntu, NOT WINDOWS. The next labs need to be carried out on Ubuntu.
   If you are using the Ubuntu Virtual Machine, the password for the default user, if needed, is "0000".
2. Once you have booted Ubuntu, open a Terminal (Ctrl+Alt+T) and create a folder to work in:

```
mkdir lab1 && cd lab1
```

3. If you are working on a LABINF PC, execute the following command:

```
export PYTHONPATH=${PYTHONPATH}:/opt/gem5-22.1-ASE/gem5/configs/
```

   If you are using the Virtual Machine:

```
export PYTHONPATH=${PYTHONPATH}:/home/vboxuser/gem5/configs/
```

   NOTE: the export command need to be done every time you spawn a new shell.

4. Write your RISC-V assembly program and save it in `program1.s` (any file name is ok, as long as it has the .s extension). You can open it by running the following command:

```
gedit program1.s
```

The assembly program **MUST HAVE THE FOLLOWING STRUCTURE**:

```
# Data section
.section .data
# Place here your program data.
# In this example, two vector of floats
# a vector of ints and a single int are defined
V1: .float 1.0, 2.0, 3.0, 4.0
```

```
V2: .float 5.0, 6.0, 7.0, 8.0
V3: .word 0, 1, 2, 3
T0: .word 0x0BADC0DE

# Code section
.section .text
# The _start label signals the entry point of your program
# DO NOT CHANGE ITS NAME.
# It must be "_start", not "start", not "main", not "start_".
# It's "_start" with a leading '_' and all lowercase letters
.globl _start
_start:
      # In the _start area, load the first byte/word of each of
      # the areas declared in the .data section
      # This is needed to load data in the cache and avoid
      # pipeline stalls later
      la x1, V1
      flw fs1, 0(x1)
      la x1, V2
      flw fs1, 0(x1)
      la x1, V3
      lw x2, 0(x1)
      la x1, T0
      lw x2, 0(x1)
Main:
      # Your code goes here
      addi x1,x0,0
End:
# exit() syscall. This is needed to end the simulation
# gracefully
      li a0, 0
      li a7, 93
      ecall
```

5. Compile using `riscv_compile`, passing `program1.s` as parameter. If you are working on a LABINF PC, you will find `riscv_compile` under `/opt/gem5-22.1-ASE/gem5_example`. The complete command will be:

```
/opt/gem5-22.1-ASE/gem5_example/riscv_compile program1.s
```

The compiled program will have the same name as the assembly file and no extension, e.g. `program1.s -> program1`

6. Copy the gem5 configuration file `gem5_config.py` from either the "Visualizer Example" folder on your Desktop (if you are using the VM) or from `/opt/gem5-22.1-ASE/gem5_example` (if you are using a LABINF PC).

```
cp /opt/gem5-22.1-ASE/gem5_example/gem5_config.py .
```

7. Simulate your program using gem5. The simulation can be carried out using the gem5_run command on LABINF PCs:

```
/opt/gem5-22.1-ASE/gem5_example/gem5_run gem5_config.py program1
program1.log
```

On the Virtual Machine:

```
gem5_run gem5_config.py program1 program1.log
```

- The `gem5_config.py` file contains the pipeline configuration. Here you can modify the operation latency and the issue latency of integer and floating point functional units (ALU, Multiplier, Divider):

```
...
INTEGER_ALU_LATENCY = 1
INTEGER_MUL_LATENCY = 1
INTEGER_DIV_LATENCY = 1
FLOAT_ALU_LATENCY = 3
FLOAT_MUL_LATENCY = 5
FLOAT_DIV_LATENCY = 5

INTEGER_ALU_ISSUE_LATENCY = 0
INTEGER_MUL_ISSUE_LATENCY = 0
INTEGER_DIV_ISSUE_LATENCY = 0
FLOAT_ALU_ISSUE_LATENCY = 0
FLOAT_MUL_ISSUE_LATENCY = 0
FLOAT_DIV_ISSUE_LATENCY = FLOAT_DIV_LATENCY
...
```

If the simulation throws an error because it cannot find the "common" module, copy the "common" folder from the same folder where you have found `gem5_config.py` into the folder where your `gem5_config.py` is (watch point 6.)

- `program1` is the program you have compiled before. If it is in a different path than the folder you are currently in, you need to provide the full path.
- `program1.log` is the output trace produced by gem5. Remember the folder you are working in, since the log file is needed for the next step.

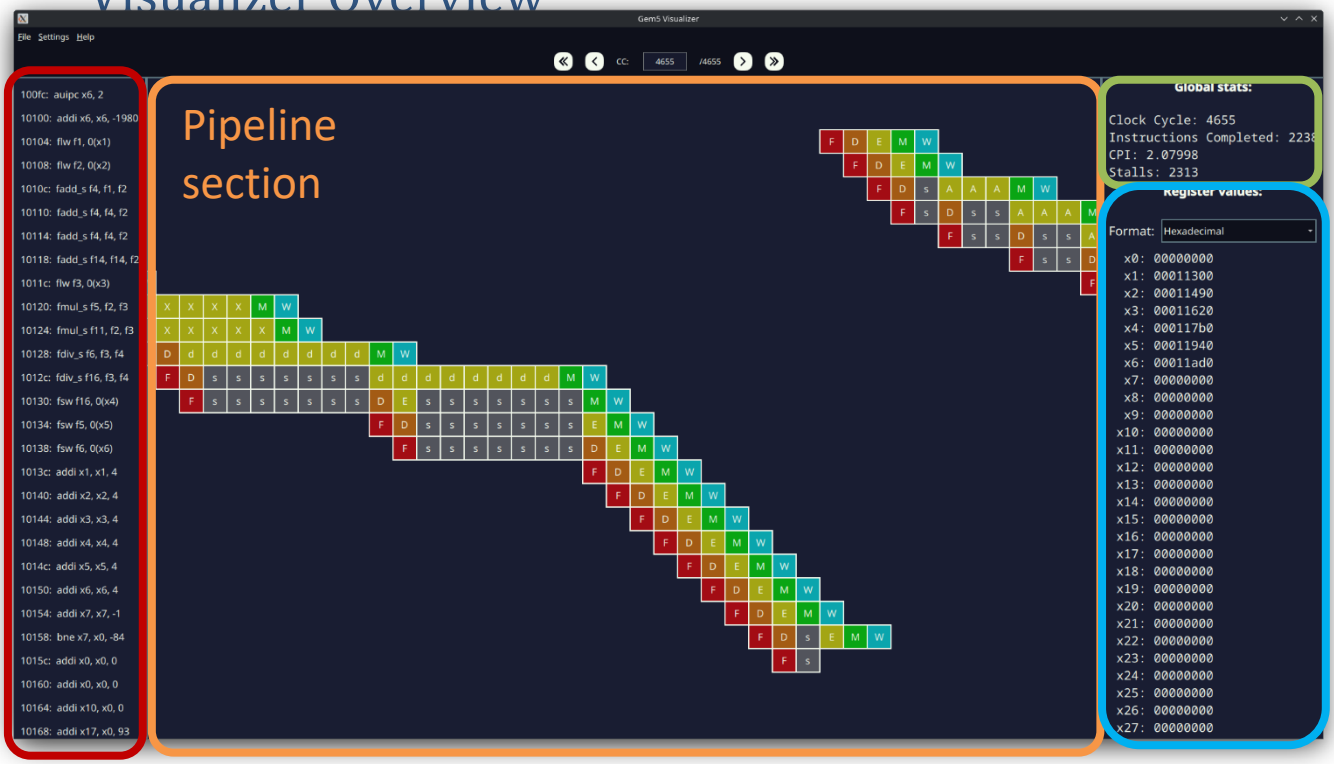
8. Open the Pipeline Visualizer. On LABINF PCs, the visualizer is available under /opt/gem5-22.1-ASE:

```
/opt/gem5-22.1-ASE/Gem5_Pipeline_Visualizer-x86_64.AppImage
```

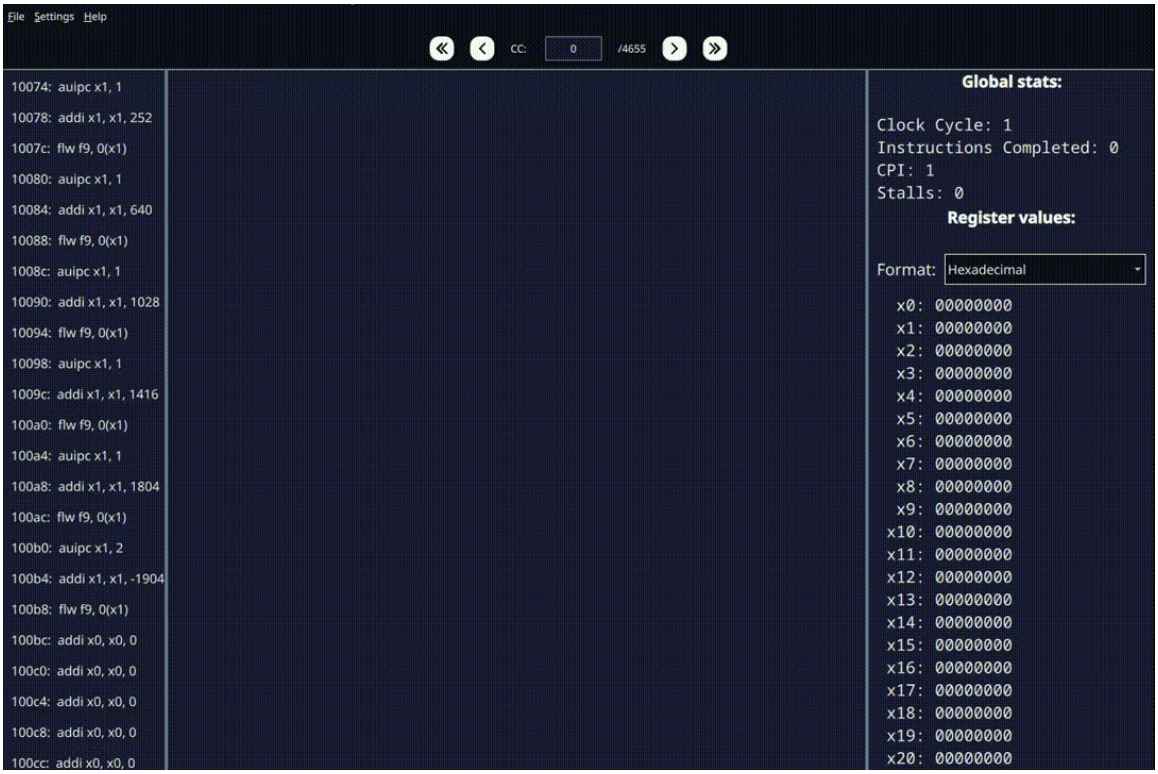
If you are using the VM, you can use the shortcut on your Desktop.

9. Click on File -> Open and open program1.log

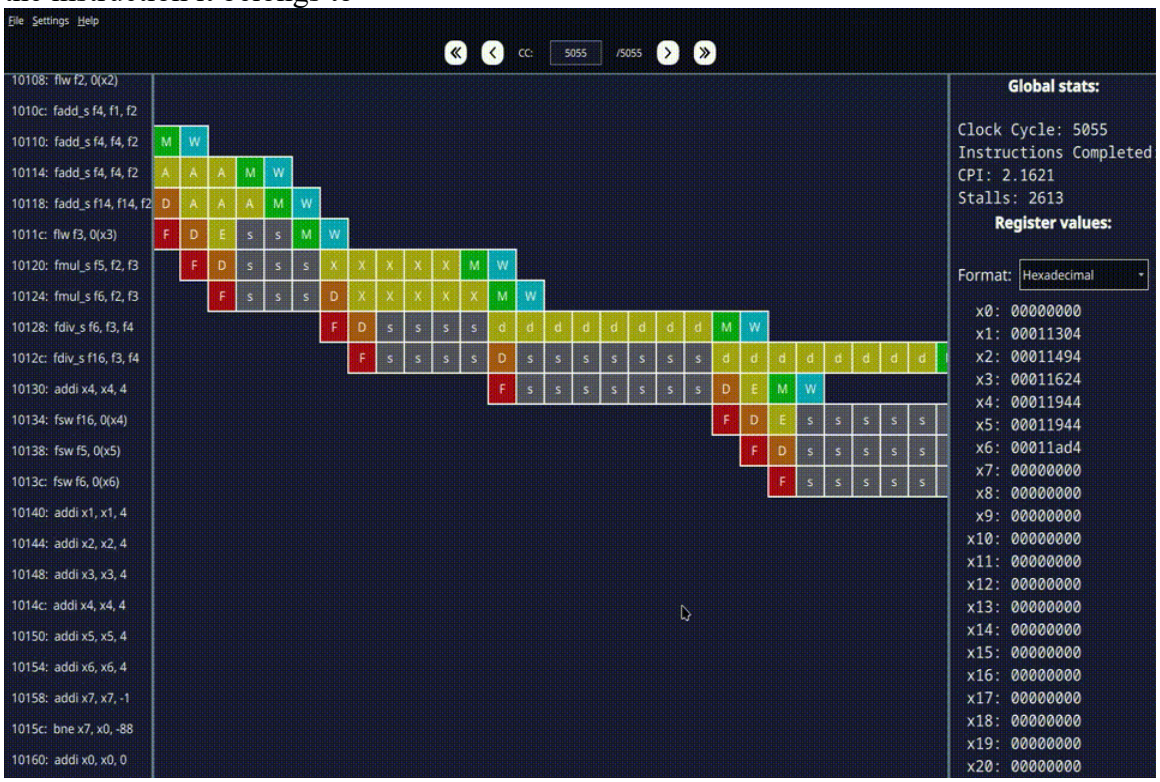
Visualizer Overview



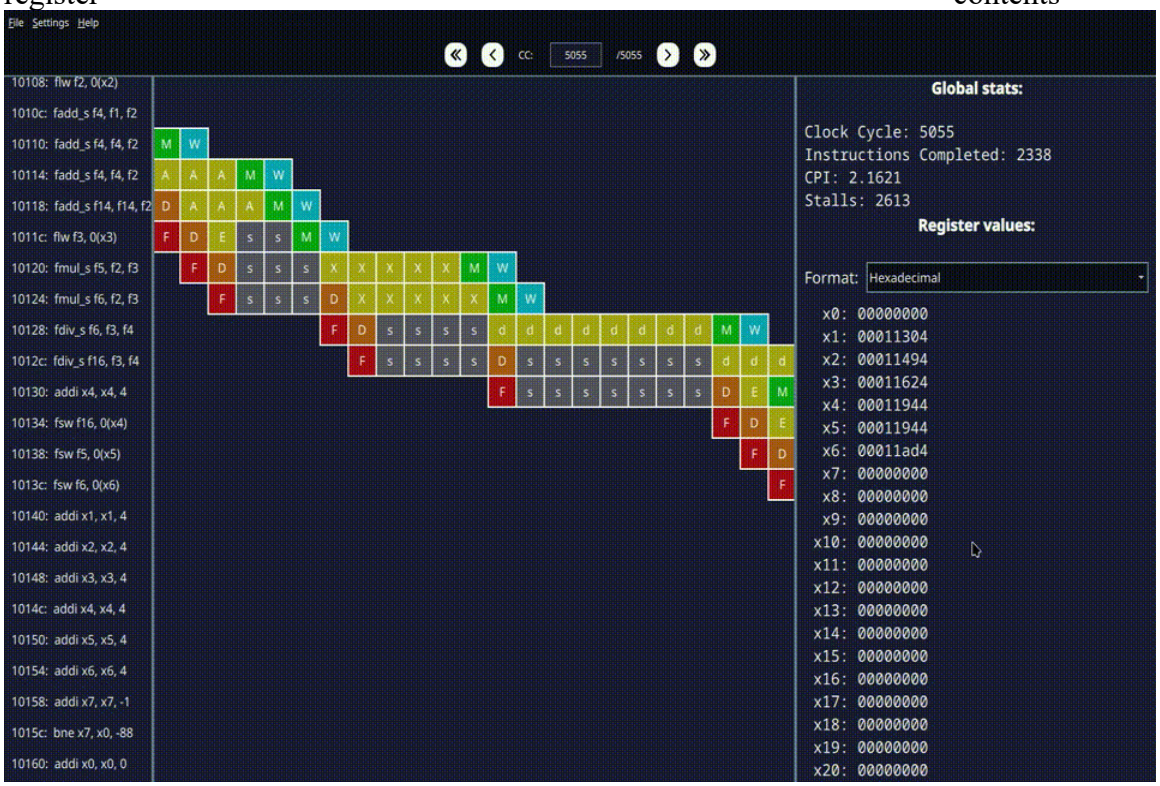
- Use the navigation buttons to advance/rewind the pipeline
  - You can fast-forward to the end, fast-rewind to the beginning. Or advance/rewind by a single clock
  - You can also choose a specific clock cycle to jump to



- Hovering your mouse over an instruction highlights its path through the pipeline. Hovering your mouse over a pipeline stage shows the clock cycle and highlights the instruction it belongs to



- Use the dropdown menu in the register section to choose the representation of register contents



- Exercise 1

Using the same flow described before, write and run a program for calculating the Fibonacci sequence and save it in `program2.s` (any file name is ok, as long as it has the .s extension). The assembly program:

```
# Data section
.section .data
# Place here your program data. In this example
# two vector of floats, a vector of ints anda single int are
defined
T0: .word 0x0BADC0DE

# Code section
.section .text
# The _start label signals the entry point of your program
# DO NOT CHANGE ITS NAME. It must be "_start", not "start",
# not "main", not "start_".
# It's "_start" with a leading underscore and
# all lowercase letters
.globl _start
_start:
# In the _start area, load the first byte/word of each of
# the areas declared in the .data section
# This is needed to load data in the cache and avoid
# pipeline stalls later
Main:
    # Initialize Fibonacci variables
    li x1, 0        # x1 = a = first Fibonacci number (0)
    li x2, 1        # x2 = b = second Fibonacci number (1)
    li x3, 21       # x3 = count = number of terms to generate
    li x4, 0        # x4 = i = loop counter

    # Loop to generate and print remaining 20 numbers
    addi x4, x4, 1  # i = 1 (start from second iteration)

fib_loop:
    beq x4, x3, End # if i == count, exit loop

    # Calculate next Fibonacci number
    add x5, x1, x2  # x5 = next = a + b

    # Update variables for next iteration
    mv x1, x2       # a = b (previous second becomes first)
    mv x2, x5       # b = next (calculated next becomes second)
     # Increment counter and continue loop
    addi x4, x4, 1  # i++
    j fib_loop      # Jump back to loop start

End:
# exit() syscall. This is needed to end the simulation
# gracefully
```

```
li a0, 0
li a7, 93
ecall
```

The above assembly code implements the following C code for calculating the Fibonacci sequence:

```c
/* Fibonacci sequence generator in C (pseudocode)
 * Generates the first 21 Fibonacci numbers using iterative approach
 */
int main() {
    int i;          // Loop counter
    int a = 0;          // First Fibonacci number
    int b = 1;          // Second Fibonacci number
    int next;           // Next Fibonacci number
    int count = 21;     // Number of terms to generate

    // Generate and print remaining numbers
    for (i = 1; i < count; i++) {
        next = a + b;   // Calculate next Fibonacci number
        a = b;          // Update: previous second becomes first
        b = next;       // Update: calculated next becomes second
    }

    return 0;
}
```

- Exercise 2

Using the same flow described before, write and run an assembly program called **program_3.s (to be delivered)** for the *RISC-V* architecture.

The program must:

1. Given two arrays of 10 8-bit integer numbers (v1,v2), check if any element of v1 is included in v2, at least once. Save the matching value in a third vector (v3).

For example:

```
v1:     .byte   2, 6, -3, 11, 9, 18, -13, 16, 5, 1
v2:     .byte   4, 2, -13, 3, 9, 9, 7, 16, 4, 7
```

The third vector will be composed as follows.

```
v3:     .byte           2, 9, -13, 16
```

2. Set three flags (flag1, flag2, flag3) to indicate three conditions:
   a. The third vector *(*v3*)* is empty. Use one 8-bit unsigned variable (flag1) to flag the condition. The variable will be equal to 1 if v3 is empty, 0 otherwise.

b. The third vector *(*v3*)* is not empty, and each element is greater than the previous one (v3[i+1]>v3[i]). In this case, use one 8-bit unsigned variable (flag2) to flag the condition. The variable will be equal to 1 if condition is satisfied, 0 otherwise.

c. The third vector *(*v3*)* is not empty, and each element is smaller than the previous one (v3[i+1]<v3[i]). In this case, use one 8-bit unsigned variable (flag3) to flag the condition. The variable will be equal to 1 if condition is satisfied, 0 otherwise.

If you see stalls when loading data at the beginning of the program, can you explain why that happens?

Gli stall iniziali compaiono perché le prime `lb` leggono dati non ancora in D-cache: c'è un cold miss, quindi l'unità MEM deve attendere che la linea arrivi dalla memoria e la pipeline si blocca con interlock; IF e ID restano fermi e nel visualizzatore gli "S" appaiono sulla riga di `auipc` solo perché il frontend è congelato mentre la `lb` aspetta; se la configurazione usa memoria a porta singola, l'accesso ai dati in MEM impedisce anche il fetch nello stesso ciclo e aggiunge stall strutturali.

Rember that after the declaration of the vectors, you were instructed to write (and adapt) few lines as described here.
Collect the clock cycles to fill the following table.

Table 1: **Program performance for the specific processor configurations**

Program	Clock cycles	Number of Instructions	Clocks per instruction (CPI)	Instructions per Clock (IPC)
`program_1`	`23`	`13`	`1.769`	`0.565`
`program_2`	`156`	`126`	`1.238`	`0.808`
`program_3`	`793`	`594`	`1.335`	`0.749`

- Exercise 3

Perform execution and CPI measurements of some benchmarks programs. Attached to the folder of this laboratory, there are the two following programs:

a) calculate_pi.s
b) insertion_sort.s

Do the same with the programs of the previous exercises:

a) program_1.s
b) program_2.s

c) program_3.s

In the initial scenario, it is assumed that the weight of the programs is the same (20%) for everyone. Assume a processor frequency of 1.75 kHz (a very old technology node).

Fill the following table assuming different scenarios:

- Scenario 1 :
  - program_1.s weights 1%
  - program_2.s weights 50%
  - program_3.s weights 13%
  - calculate_pi.s weights 25%
  - insertion_sort.s weights 11%

- Scenario 2:
  - program_1.s weights 10%
  - program_2.s weights 5%
  - program_3.s weights 50%
  - calculate_pi.s weights 10%
  - insertion_sort.s weights 25%

- Scenario 3:
  - program_1.s weights 20%
  - program_2.s weights 30%
  - program_3.s weights 1.9%
  - calculate_pi.s weights 31.4%
  - insertion_sort.s weights 16.7%

Table 2: **Processor performance for different weighted programs**

Program	Initial scenario	Scenario 1	Scenario 2	Scenario 3
calculate_pi.s	1.731 s	0.4328 s	0.1731 s	0.5435 s
insertion_sort.s	7.743 s	0.8517 s	1.9358 s	1.2931 s
program_1.s	0.013 s	0.0001 s	0.0013 s	0.0026 s
program_2.s	0.089 s	0.0445 s	0.0045 s	0.0267 s
program_3.s	0.453 s	0.0589 s	0.2265	0.0086 s
TOTAL Time (@ 1.75kHz)	10.029 s	1.3880 s	2.3412 s	1.8745 s