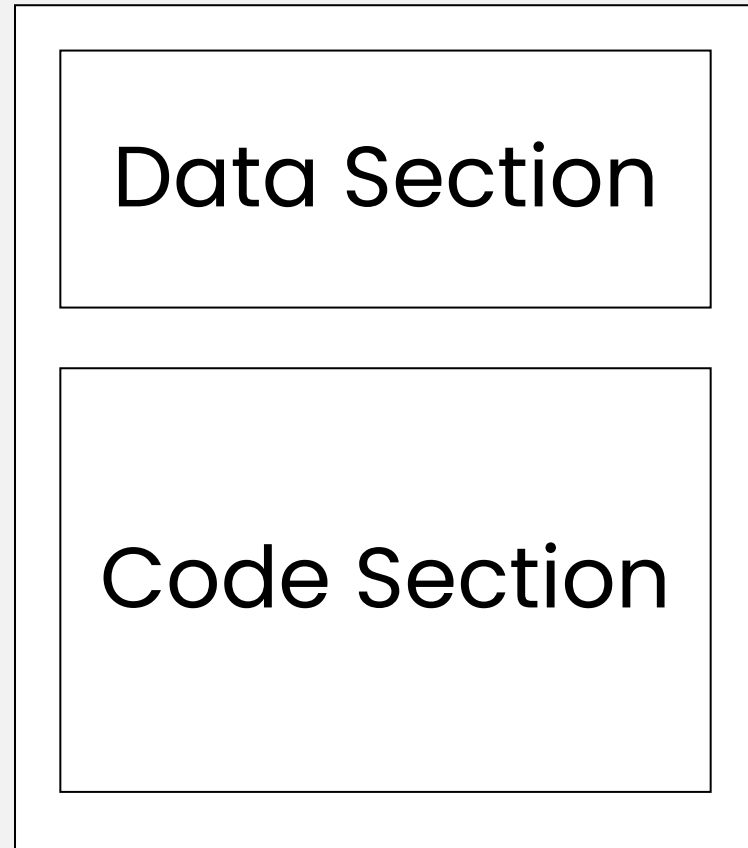


# **RISC-V Introduction**

## **gem5 and RISC-V**

Ernesto Sanchez

# ASSEMBLER PROGRAMS



Assembler program

## **Data Section**

Variables  
Constants

## **Code Section**

Program  
Routines  
Subroutines

# Data Section

```
***** RISC-V INITIAL PROGRAM*****  
#-----  
# AddVectorValues.s  
# this program adds the element in a vector  
# requires integer data in mem  
#-----
```

← Program Title

.data

← Assembler Directives

V1: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

← Constants

result: .word 0

← Variables

# Code Section

	.Code		
		.global main	
main:	addi	r1,r0,Info	*** Read value from stdin
	Jal	Input	into R1
	movi2fp	f10,r1	*** init values
	cvti2d	f0,f10	#R1 -> D0 D0..Count
	addi	r2,r0,1	register
	movi2fp	f11,r2	
	cvti2d	f2,f11	#1 -> D2 D2..result
	Movd	f4,f2	#1-> D4 D4..Constant 1
Loop:	led	f0,f4	*** Break loop if D0 = 1
	bfpt	EndL	#D0<=1 ?
	Multd	f2,f2,f0	*** Multiplication and
	subd	f0,f0,f4	next loop
	j	Loop	
			*** write result to tdout
EndL:	sd	Print,f2	
	addi	r14,r0,Print	
	trap	5	*** end

■ Assembler Directives

■ Labels

■ OPcode

■ Operators

■ Comments

# Gem5

- Gem5 is an open source system-level and microarchitectural simulator
- Supports various ISAs, including ARM, RISC-V and x86
- Can simulate a full system and offers fine-grained control over all components, including cache system, branch predictor and CPU functional units

```
gem5 version 22.1.0.0
gem5 compiled Aug 27 2025 11:38:55
gem5 started Sep 16 2025 11:44:37
gem5 executing on bernoulli, pid 3456889
command line: ./build/RISCV/gem5.debug --debug-flags=MinorGUI gem5_config.py --cpu-type Min
orCPU --caches --l1d_size 8388608 --l1i_size 8388608 --cacheline_size 512 --cpu-clock 1MHz
--sys-clock 10GHz -c test_progs/last/last

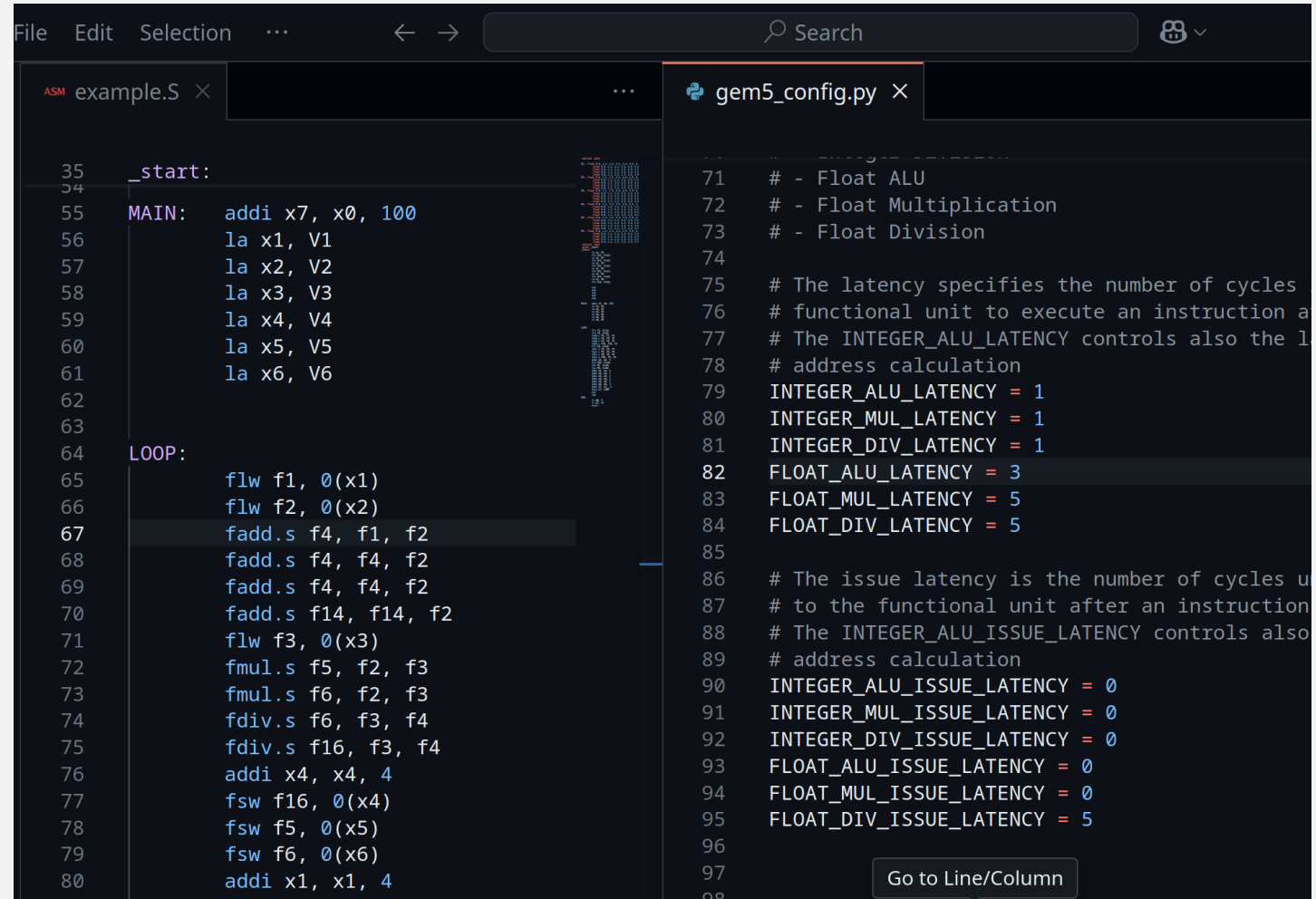
**** REAL SIMULATION ****
build/RISCV/sim/simulate.cc:192: info: Entering event queue @ 0. Starting simulation...
1000000: system.cpu.fetch1: Log4GUI: fetch1: 1000000: 0: 10074: <assembly>
2000000: system.cpu.fetch1: Log4GUI: fetch1: 2000000: 1: 10074: <assembly>
3000000: system.cpu.fetch1: Log4GUI: fetch1: 3000000: 1: 10074: <assembly>
4000000: system.cpu.fetch1: Log4GUI: fetch1: 4000000: 1: 10074: <assembly>
5000000: system.cpu.decode: Log4GUI: decode: 5000000: 0: 10074: auipc x1, 1
5000000: system.cpu.fetch1: Log4GUI: fetch1: 5000000: 0: 10078: <assembly>
6000000: system.cpu.execute: Log4GUI: execute: 6000000: 0: 10074: auipc x1, 1: 0
6000000: system.cpu.decode: Log4GUI: decode: 6000000: 0: 10078: addi x1, x1, 256
6000000: system.cpu.fetch1: Log4GUI: fetch1: 6000000: 0: 1007c: <assembly>
7000000: system.cpu.memory: Log4GUI: memory: 7000000: 0: 10074: auipc x1, 1
7000000: system.cpu.execute: Log4GUI: execute: 7000000: 0: 10078: addi x1, x1, 256: 0
7000000: system.cpu.decode: Log4GUI: decode: 7000000: 0: 1007c: flw f9, 0(x1)
7000000: system.cpu.fetch1: Log4GUI: fetch1: 7000000: 0: 10080: <assembly>
8000000: system.cpu.writeback: Log4GUI: writeback: 8000000: 0: 10074: auipc x1, 1
8000000: system.cpu.writeback: REGISTERS
8000000: global: x0=0x00000000
8000000: global: x1=0x00011074
8000000: global: x2=0x7fffff88
8000000: global: x3=0x00000000
```



# Gem5 - 2

## System definition and simulation

- Python scripts
  - Define components of the system
    - Including microarchitectural details, e.g. clock frequency, cache latencies, CPU functional units, ...
  - Configure and automate simulations
  - Collect statistics
- Executable code
  - Mainstream compilers (gcc, clang) can be used to build programs for simulation



The screenshot shows a code editor with two tabs: 'example.S' and 'gem5\_config.py'. The 'example.S' tab displays assembly code for a RISC-V program. It starts with a '\_start:' label, followed by a 'MAIN:' section where registers x1 through x6 are loaded with values from memory. Then, a 'LOOP:' section begins, containing a series of floating-point instructions: loading f1 and f2, adding f4 to f1 and f2, multiplying f5 and f6 by f2 and f3, dividing f6 by f3, and adding 4 to x4. The 'gem5\_config.py' tab shows configuration parameters for the Gem5 simulator, including latencies for Integer and Float ALU, Multiplication, and Division, and issue latencies for these units. The code is written in Python and uses comments to explain the parameters.

```
File Edit Selection ... Search
ASM example.S x
35 _start:
36
55 MAIN: addi x7, x0, 100
56      la x1, V1
57      la x2, V2
58      la x3, V3
59      la x4, V4
60      la x5, V5
61      la x6, V6
62
63
64 LOOP:
65      flw f1, 0(x1)
66      flw f2, 0(x2)
67      fadd.s f4, f1, f2
68      fadd.s f4, f4, f2
69      fadd.s f4, f4, f2
70      fadd.s f14, f14, f2
71      flw f3, 0(x3)
72      fmul.s f5, f2, f3
73      fmul.s f6, f2, f3
74      fdiv.s f6, f3, f4
75      fdiv.s f16, f3, f4
76      addi x4, x4, 4
77      fsw f16, 0(x4)
78      fsw f5, 0(x5)
79      fsw f6, 0(x6)
80      addi x1, x1, 4
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

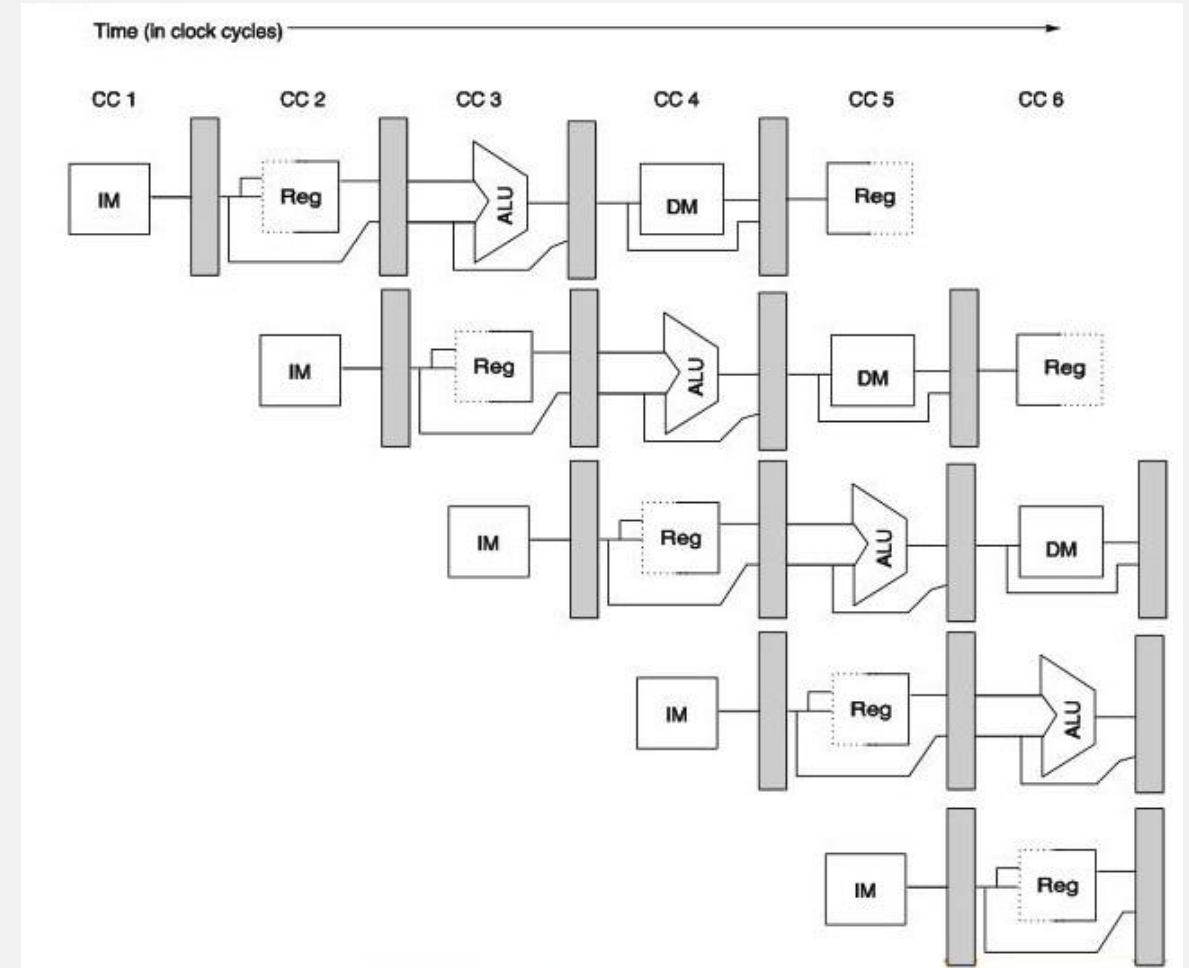
gem5_config.py x
71 # - Float ALU
72 # - Float Multiplication
73 # - Float Division
74
75 # The latency specifies the number of cycles
76 # functional unit to execute an instruction a
77 # The INTEGER_ALU_LATENCY controls also the l
78 # address calculation
79 INTEGER_ALU_LATENCY = 1
80 INTEGER_MUL_LATENCY = 1
81 INTEGER_DIV_LATENCY = 1
82 FLOAT_ALU_LATENCY = 3
83 FLOAT_MUL_LATENCY = 5
84 FLOAT_DIV_LATENCY = 5
85
86 # The issue latency is the number of cycles u
87 # to the functional unit after an instruction
88 # The INTEGER_ALU_ISSUE_LATENCY controls also
89 # address calculation
90 INTEGER_ALU_ISSUE_LATENCY = 0
91 INTEGER_MUL_ISSUE_LATENCY = 0
92 INTEGER_DIV_ISSUE_LATENCY = 0
93 FLOAT_ALU_ISSUE_LATENCY = 0
94 FLOAT_MUL_ISSUE_LATENCY = 0
95 FLOAT_DIV_ISSUE_LATENCY = 5
96
97
98
99
100
Go to Line/Column
```

# Gem5 - 3

## RISC-V 5-stage pipeline

- Custom CPU architecture for gem5 developed specifically for this course
- 5-stage pipeline: Fetch, Decode, Execute, Memory, Writeback
- Accurately mirrors the pipeline described in Hennessy-Patterson

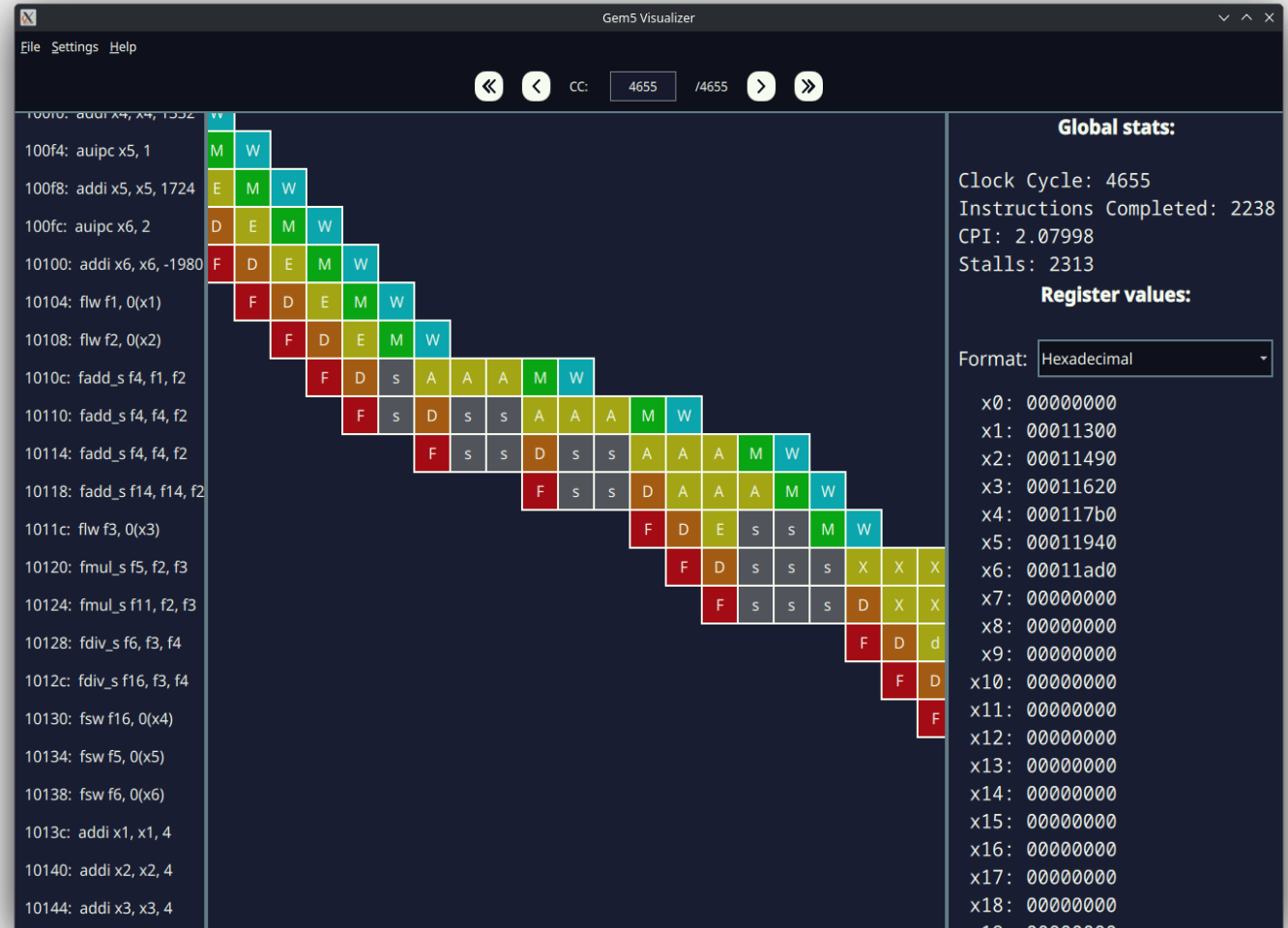
Mismatch reports are welcome!



# Gem5 Pipeline Visualizer v0.1 – 4

## Pipeline visualization

- Interactive pipeline diagrams
  - Shows the pipeline evolution during program execution.
- Reads the log file produced by Gem5 and builds an interactive pipeline diagram
  - Compilation, simulation and visualization are separate steps with separate programs!





# Gem5 Pipeline Visualizer v0.1 – 5

Instruction  
section

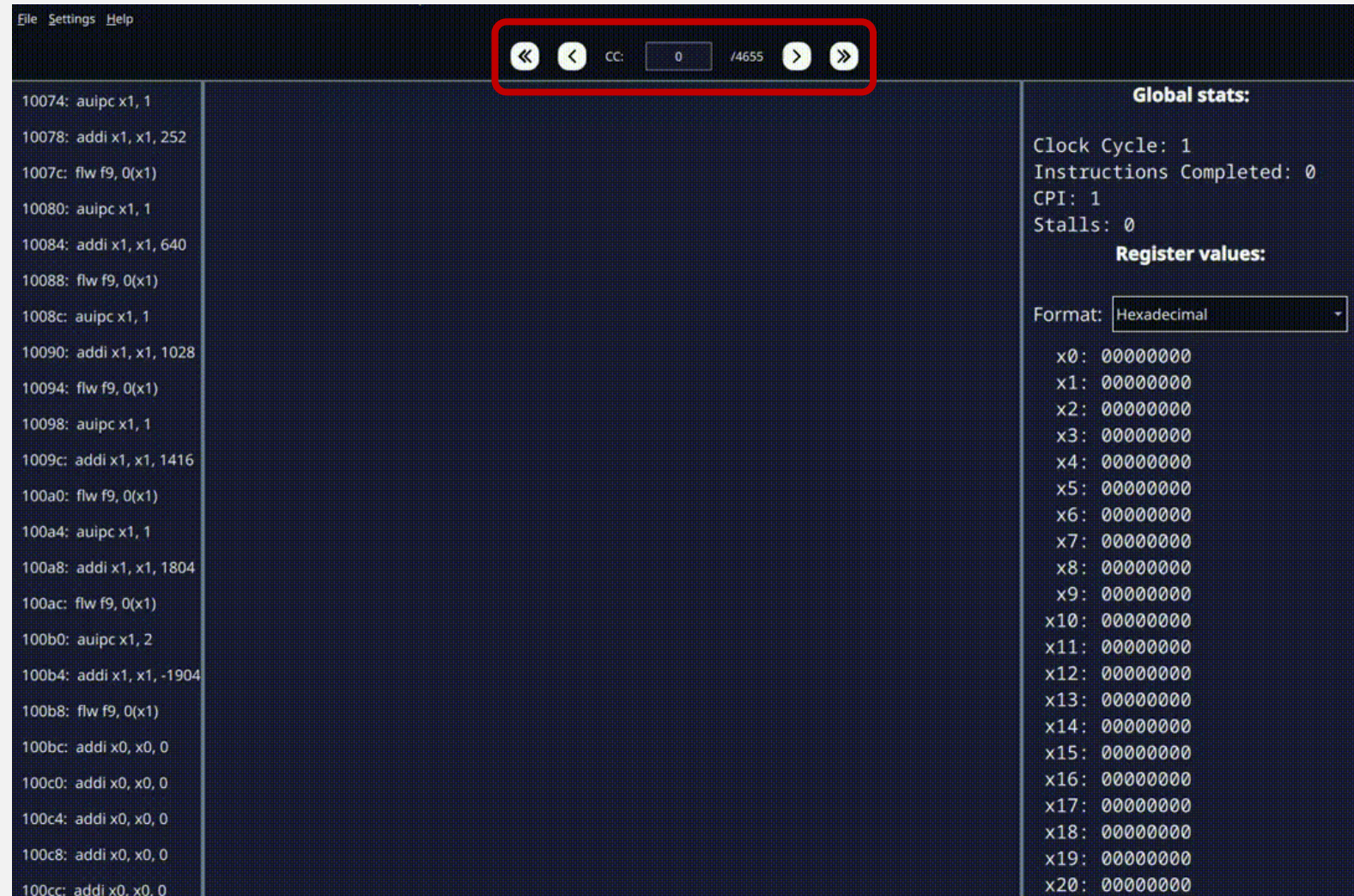


Statistics  
section

Register  
section

# Gem5 Pipeline Visualizer v0.1 – 6

- Use the navigation buttons to advance/rewind the pipeline
  - You can fast-forward to the end, fast-rewind to the beginning, or advance/rewind a single cycle
  - You can also choose a specific clock cycle to jump to
- The width of each section can be adjusted





# Gem5 Pipeline Visualizer v0.1 – 7

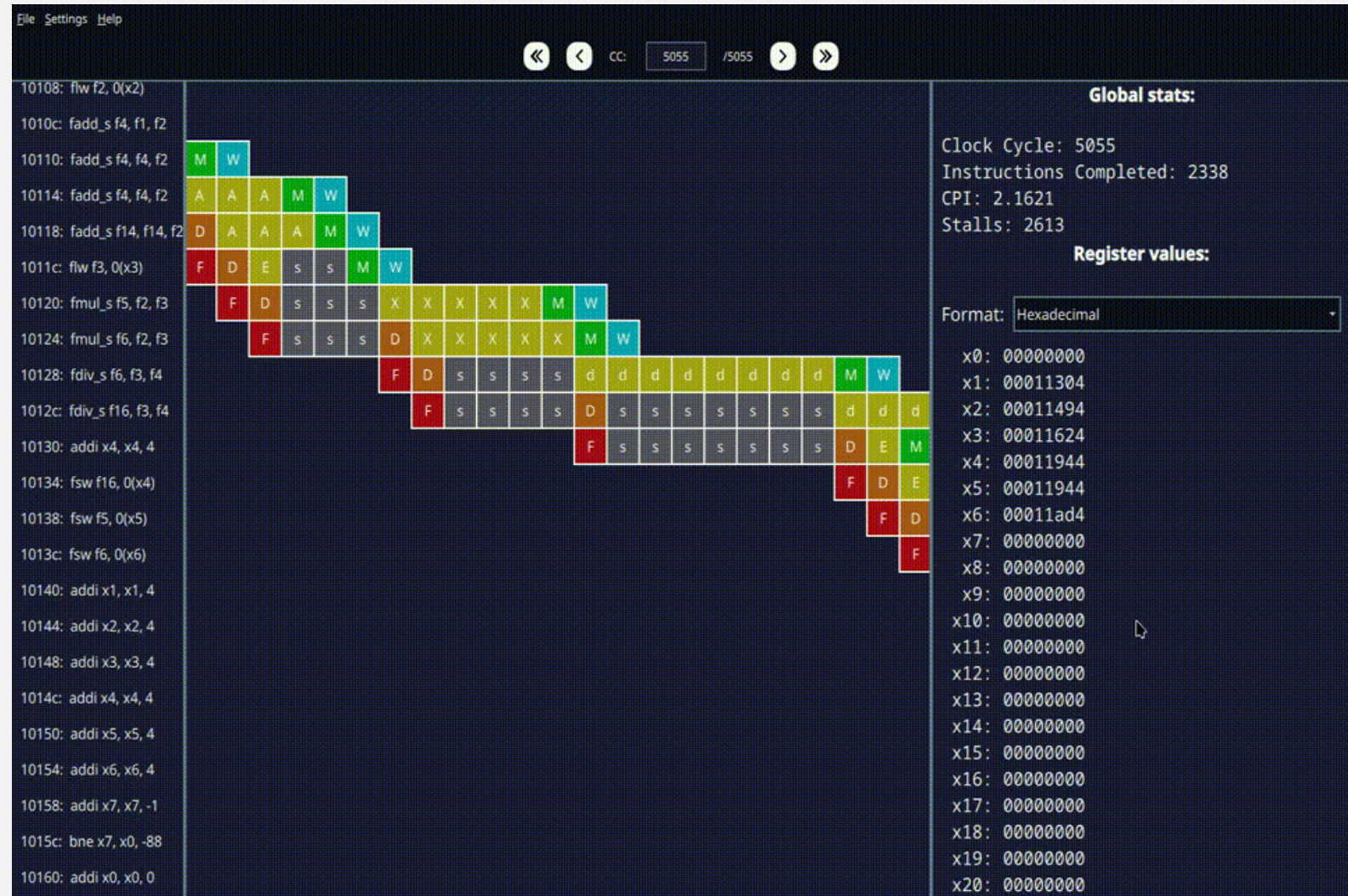
- Hovering your mouse over an instruction highlights its path through the pipeline
- Hovering your mouse over a pipeline stage shows the clock cycle and highlights the instruction it belongs to





# Gem5 Pipeline Visualizer v0.1 – 8

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



# Simulation quick guide – 9

- The required tools are available on your LABINF Ubuntu account under `/opt/gem5-22.1-ASE`
- The “gem5\_example” subfolder contains an example program (`example.s`) and compilation and simulation scripts (`riscv_compile` and `gem5_run`)

1. Open a Terminal (Ctrl+Alt+T) and run:

```
cd /opt/gem5-22.1-ASE/gem5_example
```

2. Compile the example:

```
./riscv_compile example.s
```

3. Run the simulation

```
./gem5_run gem5_config.py example test.log
```

4. Open the visualizer and open `test.log`

```
./gem5_pipeline_visualizer-x86_64.AppImage
```

# A first example

$$C = A + B$$

```
.section .data
Val_A: .word 10
Val_B: .word 20
Val_C: .word 0
.section .text
.globl _start
_start:
    la x1, Val_A
    lw x1, 0(x1)
    la x2, Val_B
    lw x2, 0(x2)
    add x3, x2, x1
    la x4, Val_C
    sw x3, 0(x4)
```

## RISC-V PROGRAM

```
.data
Val_A: dw 10
Val_B: dw 20
Val_C: dw 0

...
Main:
    mov AX, Val_A
    add AX, Val_B
    mov Val_C, AX
```

## 8086 PROGRAM

# A first example (I)

$C = A + B$

```
.section .data
Val_A: .word 10
Val_B: .word 20
Val_C: .word 0
.section .text
.globl _start
_start:
    la x1, Val_A
    lw x1, 0(x1)
    la x2, Val_B
    lw x2, 0(x2)
    add x3, x2, x1
    la x4, Val_C
    sw x3, 0(x4)
```

**RISC-V PROGRAM**

Code Analysis	
# instructions	7 → 10
Code size [bytes]	40
Execution time [C.C.]	15

# A first example (II)

$$C = A + B$$

Code Analysis	
# Instructions	3
Code size [bytes]	8
Execution time [C.C.]	33

```
.data
Val_A: dw 10
Val_B: dw 20
Val_C: dw 0

...
Main:
    mov AX, Val_A
    add AX, Val_B
    mov Val_C, AX
```

## 8086 PROGRAM



# A 2<sup>nd</sup> example

```
/* ***** */
/* Sum of two vectors */
/* ***** */

#include <stdio.h>

const int V1[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int V2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result[10];

int main() {
    for (int i = 0; i < 10; i++) {
        result[i] = V1[i] + V2[i];
    }
}
```

# A 2<sup>nd</sup> example

```
***** RISC-V INITIAL PROGRAM *****
#-----
# AddTwoVectors.s
# this program adds two vectors and stores
# the result in another vector
#-----

        .section .data
V1:      .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
V2:      .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
result:  .space 40

        .section .text
# This prologue ensures that vectors are in
# the cache when we need them, otherwise
# the pipeline will stall at first access
        .globl _start
_start:
        la x1, V1
        lw x2, 0(x1)
        la x1, V2
        lw x2, 0(x1)
        nop
        nop
```

Since we don't have a main, the `_start` symbol indicates where the beginning of the program is. **Always use the `_start` label, not `start`, not `START`, not `main`!**

```
        nop
        nop
main:
        la x1, V1
        la x2, V2
        la x3, result
        addi x4, x0, 10 # iteration counter

cycle:
        lw x5, 0(x1)
        lw x6, 0(x2)
        add x7, x5, x6
        sw x7, 0(x3)
        addi x1, x1, 4
        addi x2, x2, 4
        addi x3, x3, 4
        addi x4, x4, -1
        bnez x4, cycle
        nop

END_of_PROGRAM:
        li a0, 0 # Syscall parameter
        li a7, 93 # exit() syscall number
        ecall
```

Use the `exit()` syscall to stop the gem5 simulation

# References

- <https://github.com/cad-polito-it/gem5>
- [https://github.com/cad-polito-it/gem5\\_visualizer](https://github.com/cad-polito-it/gem5_visualizer)
- [https://github.com/cad-polito-it/gem5\\_visualizer\\_example](https://github.com/cad-polito-it/gem5_visualizer_example)
- [https://www.gem5.org/documentation/general\\_docs/building](https://www.gem5.org/documentation/general_docs/building)