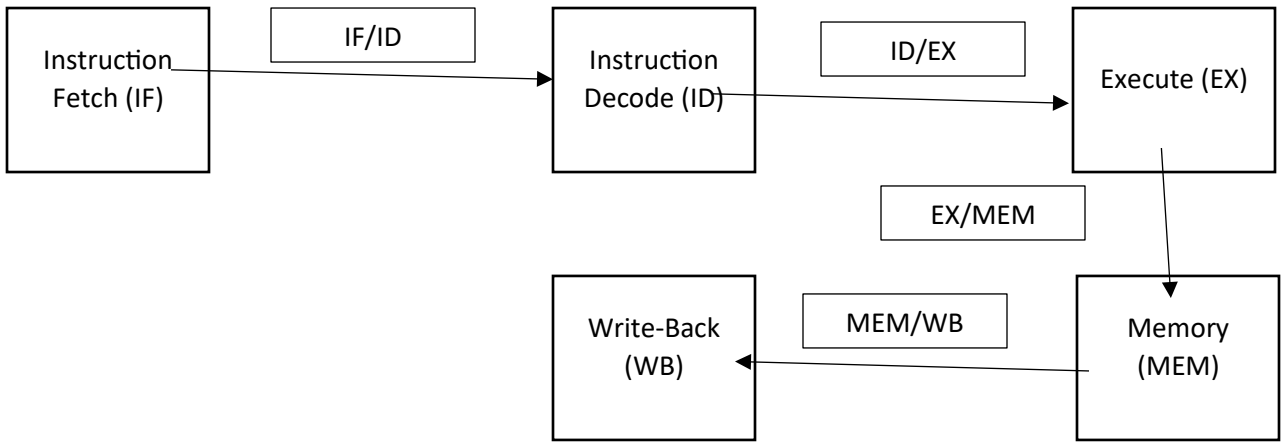
Processor Implementation – ELEC2204

Kithmal Amarasinghe
akca1e22@soton.ac.uk
Personal Tutor: Igor Golosnoy

1. Design

1.1 – Implemented Processor Diagram



- The IF segment of the processor read instruction text from instruction_memory[pc] and then thereafter incremented 'pc'. Raw text was stored into the IF/ID register
- The ID segment of the processor used the parse_instruction function to turn the read text into an actual instruction in the form of (opcode, rs, tr, red/imm). If a hazard were detected this is where it would be stalled (ID/EX) otherwise it passes, and the operand are sent to the ID/Ex register.
- The EX segment of the processor uses detect_fowarding sources to determine whether to take operands form the register file, EX/MEM result or MEM/WB result. When the operand was considering ALU operations such as add, and sub can be done, and the result is stored in the EX/MEM register.
- The MEM segment of the processor used read or write data_memory[addr//4] to implement the 'lw' and 'sw' respectively. The next instruction is stored in the MEM/WB register.
- Finally, the WB segment write a register is the given conditions are met as well as increments the global variable instr_executed
- c oscillators is a key area of study in the field of nonlinear dynamics. It ties practical engineering applications to theoretical physics. Numerous chances exist for learning about and understanding complicated behaviours that appear random but are deterministic in nature in chaotic systems. Their deterministic character and sensitivity to initial conditions set them apart. Chaotic oscillators are being researched in a variety of scientific and industrial disciplines because of their unique properties, such as fractal structures and non-repeating patterns.

1.2 – Registers, Memory Layout and Instruction Set

Register	Purpose
$0	Hardwired zero (always 0)
$1	Base pointer into data memory
$2	Loop counter / general-purpose
$3	ALU result (e.g. computed square)
$4	ALU increment (e.g. odd number)
$5	Reserved for limit/immediate values
$6	Reserved for comparisons/temps
$7–$31	General-purpose

The memory layout consists of 1024 words. The addressing for this memory layout is word-aligned. The effective byte address is calculated by dividing by 4 which gives us the word index

The instruction memory is stored as a list of text lines in instruction_memory. The PC starts at 0 and is incremented by the instruction fetch (IF) segment of the processor. Any found hazards are resolved at load time and are not used in the program unless branching is involved.

Supported instruction sets include the following:

Format	Use Case
R-type	Add/sub/and/or/slt rd, rs, rt
I-type	Addi/slti rt, rs, imm Lw/sw rt, offset(rs)
Branch	Bne rs, rt, label

2. Functionality

The instruction-fetch stage is responsible for reading the next instruction from the instruction memory and then advancing the programme counter (PC). In the reference implementation, the PC is an integer that starts at 0. During every cycle's IF stage, the simulator first checks whether the current PC value is still within the bounds of the instruction-memory list (pc < len(instruction_memory)). If it is, the instruction at that index is placed into the pipeline['IF_ID'] register, a log entry of the form "Fetched instruction: …" is appended, and the PC is incremented by one so that the next cycle will examine the following instruction. When the PC eventually moves beyond the final entry in instruction_memory, the IF stage inserts a neutral operation (NOP) by setting pipeline['IF_ID'] to None; this prevents the downstream stages from attempting to decode or execute non-existent instructions. Two principal data structures underpin this stage. The first is instruction_memory, a list of text lines that has already been populated by the load_program() routine. The second is the IF/ID register itself, which temporarily stores the raw instruction text ready for the Decode stage.

In the decode stage the simulator converts the raw instruction text just fetched into a structured form, checks for load–use data hazards and gets every control signal or operand ready for the subsequent pipeline stages. The work begins with parse_instruction(line), which splits the source line on spaces or commas, and returns a whose 'opcode' key. The remainder of the raw data processed depends on the instruction type. An R-type instruction carries the fields rd, rs, rt. An I-type arithmetic instruction such as addi or slti contains rt, rs, imm. Finally, memory instruction (lw or sw) uses rt, rs, imm, with imm representing the byte offset. If the line is ever blank or unrecognised, the routine yields a NOP represented by {'opcode': 'nop'} which essentially stalls the whole system. Straight after parsing, the stage calls detect_load_use_hazard(ID_instr, EX_instr). If that function reports that the instruction currently resident in EX is a load targeting a register that the just-decoded instruction wants as a source, the simulator treats this as a load–use hazard. It writes a relevant message to the log, clears pipeline['ID_EX'], pipeline['EX_MEM'], and pipeline['MEM_WB'] which stalls the system.

The execute stage is where the datapath does its arithmetic or logical work, calculates effective addresses, and determines the flow of this pipelined system. The forwarding logic begins with detect_forwarding_sources(ID_instr), which looks first into the EX/MEM register. If that latch is to write the very register the current instruction needs as a source, the corresponding forwarding selector (forwardA or forwardB) is set to 'EX'. If this check fails, the function inspects the MEM/WB register. If it will supply the desired register, the selector is set to 'MEM'. Once the selectors are known, apply_forwarding(instr, forwardA, forwardB) fetches operand values either straight from the register file or from pipeline['EX_MEM']['result'] or pipeline['MEM_WB']['result']. With operands in hand, the ALU executes the operation indicated by the opcode. R-type instructions may be added, sub, and, or**,** or **slt**; I-type arithmetic instructions are **addi** and **slti**. For a memory access, the stage computes the effective address using the word aligned addressing techniques. From a pipeline perspective, the EX stage processes the data held in pipeline['ID_EX'] together with any forwarded operands. It produces a form of data in pipeline['EX_MEM'] containing the fields result, addr, val, and rd, ready for the memory stage to pick up next cycle.

The memory stage is responsible for interacting with the data memory whenever a load word (lw) or store word (sw) instruction reaches it. The operations take place using byte address called addr because the data_memory structure is organised as an array of 32-bit words. As a result, the simulator indexes it with addr // 4, effectively converting the byte address into a word address. If the instruction is a load, the processor performs a set of instructions which placed the fetched word into mem['result'] so that the following write-back stage can forward it to the register file. If, on the other hand, the instruction is a store, it executes a set of instructions copying the value previously captured in mem['val'] to the calculated location in memory. Whichever action is taken, the stage finally passes the now-updated instruction record on to the next latch by assigning it to pipeline['MEM_WB'], ready for the write-back stage in the subsequent clock cycle.

The final stage of the pipeline returns results to the architectural state. At the start of each cycle the simulator examines pipeline['MEM_WB']. If the entry is not a NOP and the processed data contains a valid destination register (rd), it executes a set of instructions which copies either the ALU outcome or the word fetched from memory into the register file and bumping the instruction-completed counter. Most importantly, write-back is performed before the next cycle's fetch and decode stages begin, so the pipeline always commits results in strict programme order, preserving the appearance of sequential execution even though multiple instructions are in being processed simultaneously.

A summary of each clock cycle is captured by log_pipeline_state(log). The routine records the current cycle number, the contents of all four pipeline registers IF/ID, ID/EX, EX/MEM, and MEM/WB marking any empty stage as NOP. The contents of the first eight general-purpose registers can be seen here as well. It also notes the cumulative value of instr_executed, which rises every time the write-back stage successfully commits an instruction. Once simulation has run its course, the logger depicts the number of instructions executed to to the logger text file. This gives us a summary of the overall throughput. Any testing done was build unittest.TestCase which makes sure that aspects of the simulator behave like it was intended to. This will elaborated on in upcoming sections.

The simulator is organised in modular approach so that every pipeline stage keeps to its own section of code. Inside pipeline_step() each stage's logic sits in a self-contained block, making the flow of control crystal clear and the boundaries between responsibilities unambiguous. All the other functions and routines such as parse_instruction, the hazard detectors and the forwarding selector can be exercised independently in unit tests without any hidden dependencies. A single call to reset() wipes every global structure, giving each test-run a clean slate and ensuring results are fully repeatable.

3. Basic Testing

The reliability and fucntionality are tested through a series of functions which are going to be explained now. It begins with instruction parsing. The test case presents parse_instruction() with a representative mix of R-type, I-type, and load/store functions as well as blank lines, in the format of opcode, rs, rt, rd, and imm are extracted exactly as required for each format. The next step is hazard detection which is used to verify load–use checks, the test harness feeds detect_load_use_hazard() with pairs of instructions in which the ID stage either does or does not read the destination register of a preceding load, confirming that the function raises a hazard only in the true-positive cases. Forwarding logic is exercised separately via data which is placed in the EX_MEM and MEM_WB is stored which results in forwardA and forwardB selections being compared against the full matrix of expected outcomes. The correctness of forwarding application is then checked. The register file and the registers in between processors components are initialised with distinct test values. Apply_forwarding() is run in each of its three modes which include, normal register read, EX forwarding, and MEM forwarding. The operands it returns are checked against the known ground truth for every configuration. Finally, the tests turn to ALU operations. The execute stage does the required steps for either a R-type or I-type instruction. After execution, the result captured in the pipeline register is compared with the hand-calculated answer, confirming that every arithmetic and logical path produces the right output. Together these focused tests ensure each core module behaves flawlessly before larger, end-to-end scenarios are attempted.

The integration test-bench exercises the whole datapath, clock by clock, to prove that all the individual stages really do work together under realistic traffic. The testing started addi $1, $0, 5 and finishes with lw $4, 0($0). The simulator is allowed to run for twenty cycles, after which the main function asserts that both the register file and the data-memory array contain exactly the values calculated by the reference solution. The following test lines lw $1, 0($0), addi $2, $1, 1 were made to initiate a stall because the addi needs the word that is still being fetched by the preceding lw, the pipeline must recognise a load–use hazard, insert a bubble, and delay the dependent instruction by one tick. The test therefore scans the cycle log for the message the relevant message being output and inspects the IF/ID, ID/EX and EX/MEM registers to confirm that a bubble was indeed inserted in the correct cycle. Finally, a forwarding scenario verifies that the data forwarding network eliminates unnecessary stalls is tested with the following lines of code. add $1, $2, $3 add $4, $1, $5 which in theory should forward without stalling. Here the second add should receive $1 directly from the EX/MEM register, allowing it to proceed on the very next cycle. The test checks that no "Stalling" entry appears in the log and that register $4 holds the correct result after exactly two cycles, thereby demonstrating that forwarding paths are functioning as intended.

During verification the test-bench inspects the log produced by the simulator and checks that every cycle's entry is complete. Each printed output must list the raw instruction text currently in flight, the contents of all four pipeline latches, the values of registers 0 through 7, and the running total of instructions executed so far. Once execution halts the harness looks for the concluding line and asserts that whatever the final number of instruction is exactly equal to the number of instructions the write-back stage has committed, guaranteeing that the log provides an accurate and comprehensive record of pipeline activity.

4. Showcase Testing

```
# Initialise registers and constants
    addi  $1, $0, 0      # $1 ← address of first output (start at 0)
    addi  $2, $0, 0      # $2 ← n, our loop counter (start at 0)
    addi  $3, $0, 0      # $3 ← current square value (0² = 0)
    addi  $4, $0, 1      # $4 ← the "odd increment" (first odd number = 1)
    addi  $5, $0, 201    # $5 ← loop limit (we want squares up to n=200)

  loop:
    sw    $3, 0($1)      # store the current square into data memory
    addi  $1, $1, 4      # move to the next memory word (advance by 4 bytes)
    add   $3, $3, $4     # compute next square: square += odd_increment
    addi  $4, $4, 2      # update odd_increment for the next loop (+=2)

    addi  $2, $2, 1      # increment n by 1
    slti  $6, $2, 201    # set $6=1 if n<201, otherwise $6=0
    bne   $6, $0, loop   # if n is still ≤200, repeat

  end:
    nop                  # no-op to mark the end of the programme
```

The above program written in pseudocode implements the idea that $n^2 = (n-1)^2 + (2n-1)$ so that each new square is computed by adding the next odd number. The following table expands on the registers.

Register	**Purpose**
$1	Base Address (Bytes)
$2	Loop counter
$3	Current Square
$4	Current odd increment
$5	Loop limit

$6	Comparison results for branch

5. Conclusion

Over the course of this project, the development of a fully functional five-stage pipelined processor simulator in Python has progressed. It correctly implements instruction fetch, decode (with load–use hazard detection and stalling), execute (with full ALU support for R- and I-types), memory access and write-back. A forwarding unit with a sound testing system behind it, eliminates unnecessary bubbles, ensuring that dependent instructions proceed without delay wherever possible. The development of the testing structures held within unittest covered testing criteria such as parsing, hazard detection, forwarding logic, ALU operations and end-to-end instruction sequences which is vital to the production of a processor.

If a reflection standpoint were to be taken, several clear avenues still exist for extending the simulator beyond its current five-stage, data-hazard-aware baseline. The most important is most likely the implementation of control-hazard support. Implementation of branch of jump instructions such as bne, beq and j would require the logic to flish or stall the pipeline when the control flow changes. Once this approach has been implemented, branches can be used within this simulator. Furthermore, the implementation of an explicit control unit is lacking. This lack forced stager stages/components of the processor to pick up more work than required whereas if a control unit were present, it would be more efficient and would accurately reflect a hardware control path. Additionally, the development can be extended to logging more statistics as well. This includes the number of bubbles inserted and forwarding events which would aid architectural evaluation. Finally, an implementation for future work could be improving on the current instruction set. This could be done in terms of adding shift operations, logical immediate or simple multiplication via shift-add loops to broaden the applicability without introducing multiplication and division hardware specifically.

Appendix

Below is the depiction of the program used to run simulator:

```
addi $1, $0, 5
addi $2, $0, 10
add $3, $1, $2
sw $3, 0($0)
lw $4, 0($0)
```

Below is the depiction of logger:

```
Cycle 1
Fetched instruction: addi $1, $0, 5
Pipeline State:
  IF_ID: addi $1, $0, 5
  ID_EX: NOP
  EX_MEM: NOP
  MEM_WB: NOP
  Registers [0□7]: [0, 0, 0, 0, 0, 0, 0, 0]
  Instructions executed so far: 0

Cycle 2
Fetched instruction: addi $2, $0, 10
Pipeline State:
  IF_ID: addi $2, $0, 10
  ID_EX: {'opcode': 'addi', 'rt': 1, 'rs': 0, 'imm': 5}
  EX_MEM: NOP
  MEM_WB: NOP
  Registers [0□7]: [0, 0, 0, 0, 0, 0, 0, 0]
  Instructions executed so far: 0
```

Cycle 3

Fetched instruction: add $3, $1, $2

Pipeline State:

IF_ID: add $3, $1, $2

ID_EX: {'opcode': 'addi', 'rt': 2, 'rs': 0, 'imm': 10}

EX_MEM: {'opcode': 'addi', 'rt': 1, 'rs': 0, 'imm': 5, 'result': 5, 'rd': 1}

MEM_WB: NOP

Registers [0□7]: [0, 0, 0, 0, 0, 0, 0, 0]

Instructions executed so far: 0

Cycle 4

Fetched instruction: sw $3, 0($0)

Pipeline State:

IF_ID: sw $3, 0($0)

ID_EX: {'opcode': 'add', 'rd': 3, 'rs': 1, 'rt': 2}

EX_MEM: {'opcode': 'addi', 'rt': 2, 'rs': 0, 'imm': 10, 'result': 10, 'rd': 2}

MEM_WB: {'opcode': 'addi', 'rt': 1, 'rs': 0, 'imm': 5, 'result': 5, 'rd': 1}

Registers [0□7]: [0, 0, 0, 0, 0, 0, 0, 0]

Instructions executed so far: 0

Cycle 5

Fetched instruction: lw $4, 0($0)

Pipeline State:

IF_ID: lw $4, 0($0)

ID_EX: {'opcode': 'sw', 'rt': 3, 'rs': 0, 'imm': 0}

EX_MEM: {'opcode': 'add', 'rd': 3, 'rs': 1, 'rt': 2, 'result': 15}

MEM_WB: {'opcode': 'addi', 'rt': 2, 'rs': 0, 'imm': 10, 'result': 10, 'rd': 2}

Registers [0□7]: [0, 5, 0, 0, 0, 0, 0, 0]

Instructions executed so far: 1

Cycle 6

Pipeline State:

IF_ID: NOP

ID_EX: {'opcode': 'lw', 'rt': 4, 'rs': 0, 'imm': 0}

EX_MEM: {'opcode': 'sw', 'rt': 3, 'rs': 0, 'imm': 0, 'addr': 0, 'val': 15}

MEM_WB: {'opcode': 'add', 'rd': 3, 'rs': 1, 'rt': 2, 'result': 15}

Registers [0□7]: [0, 5, 10, 0, 0, 0, 0, 0]

Instructions executed so far: 2

Cycle 7

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: {'opcode': 'lw', 'rt': 4, 'rs': 0, 'imm': 0, 'addr': 0, 'rd': 4}

MEM_WB: {'opcode': 'sw', 'rt': 3, 'rs': 0, 'imm': 0, 'addr': 0, 'val': 15}

Registers [0□7]: [0, 5, 10, 15, 0, 0, 0, 0]

Instructions executed so far: 3

Cycle 8

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: {'opcode': 'lw', 'rt': 4, 'rs': 0, 'imm': 0, 'addr': 0, 'rd': 4, 'result': 15}

Registers [0□7]: [0, 5, 10, 15, 0, 0, 0, 0]

Instructions executed so far: 4

Cycle 9

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 10

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 11

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 12

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 13

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 14

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 15

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 16

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 17

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 18

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 19

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 20

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 21

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 22

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 23

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 24

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 25

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 26

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 27

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 28

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 29

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Cycle 30

Pipeline State:

IF_ID: NOP

ID_EX: NOP

EX_MEM: NOP

MEM_WB: NOP

Registers [0□7]: [0, 5, 10, 15, 15, 0, 0, 0]

Instructions executed so far: 5

Total instructions executed: 5