# Session 8: Performance and Microarchitecure

## 1 Introduction

The goal of this exercise session is to introduce you to some microarchitecure concepts and low level optimization. Learning about these optimizations will not only make you a better programmer, but will also give you more insight in to the wonderful low-level world and enhance your reasoning skills about it. You should read the book from section 4.1 to 4.9 to get a better grasp on processor design and architecture.

## 2 Ripes

To help you solve and reason about the upcoming exercises, it is advised to install the Ripes RISC-V simulator from `https://github.com/mortbopet/Ripes/releases/latest`. There is support for Windows, Mac and Linux. On Ubuntu, make the .AppImage executable using the command `chmod +x <ripes-filename>` to run the simulator.

Using Ripes, you can simulate four different processors: a single cycle RISC-V and three variants of a 5-stage pipeline. It also has a great cache simulator which can help you better understand what was learned in the caching session.

All programs presented in this session can be executed cycle per cycle using Ripes.

## 3 Microarchitecture and Performance

In this exercise we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapaths *a* and *b* have the following latencies:

| | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|
| *a* | 300ps | 400ps | 350ps | 500ps | 100ps |
| *b* | 200ps | 150ps | 120ps | 190ps | 140ps |

**Exercise 3.1.** *What is the clock cycle time in a pipelined and single-cycle non-pipelined processor?*

**Solution:**

| | Pipelined | Single-cycle |
|---|---|---|
| *a* | 500ps | 1650ps |
| *b* | 200ps | 800ps |

**Exercise 3.2.** *What is the total latency of a* `lw` *instruction in a pipelined and single-cycle non-pipelined processor?*

**Solution:**

| | Pipelined | Single-cycle |
|---|---|---|
| *a* | 2500ps | 1650ps |
| *b* | 1000ps | 800ps |

**Exercise 3.3.** *If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?*

**Solution:**

| | Stage to split | New clock cycle time |
|---|---|---|
| *a* | MEM | 400ps |
| *b* | IF | 190ps |

## 4 Microarchitecture and Performance 2

Consider a processor where the individual instruction fetch, decode, execute, memory, and writeback stages in the datapath have the following latencies:

| IF | ID | EX | MEM | WB |
|---|---|---|---|---|
| 150ps | 50ps | 200ps | 100ps | 100ps |

We assume a classic RISC instruction set where all 5 stages are needed for loads (**lw**), but the writeback **WB** stage is not necessarily needed for stores (**sw**) and the memory **MEM** stage is not necessarily needed for register (R-format) instructions.

Instead of a single-cycle organization, we can use a multi-cycle organization where each instruction takes multiple clock cycles but one instruction finishes before another is fetched. In this organization, an instruction only goes through stages it actually needs (e.g. `sw` only takes four clock cycles because it does not need the WB stage). Compare clock cycle times and execution times with single-cycle, multi-cycle, and pipelined organization.

**Exercise 4.1.** *Calculate the clock cycle time for a single-cycle, multi-cycle, and pipelined implementation of this datapath.*

**Solution:**

- single-cycle: minimum 600ps
- multi-cycle: minimum 200ps
- pipeline: minimum 200ps

**Exercise 4.2.** *What are the best and worst case total latencies for an instruction in each of the 3 designs (single-cycle, multi-cycle, pipelined)? Note: express instruction latency both in number of cycles as well as in total time (ps). Based on these numbers, which design would you prefer? Explain.*

**Solution:**

- single-cycle: always the same: 1 cycle (600ps)
- multi-cycle: best case (R-format) = 4 cycles (4*200=800ps), worst case = 5 cycles (5*200=1000ps)
- pipeline: always the same: 5 cycles (5*200=1000ps)

While single/multi-cycle designs outperform in latency, pipelined designs will easily outperform in instruction throughput (!)

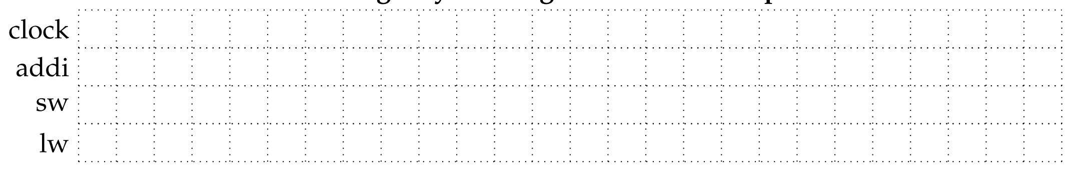
---

**Exercise 4.3.** *Consider the following RISC-V program:*

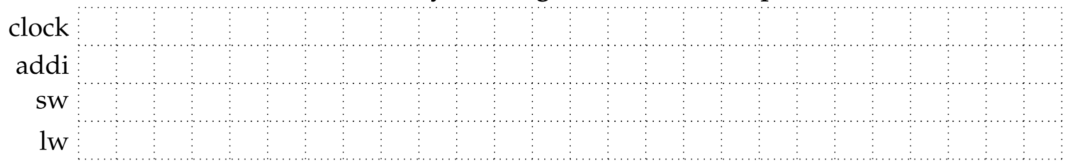
```
addi t0, zero, 10
sw zero, 4(t1)
lw t2, 10(t3)
```

*For each of the 3 CPU designs (single-cycle, multi-cycle, pipelined), we provide a grid below where the horizontal axis represents time (every cell is 100 ps), and the vertical axis lists the instruction stream. First draw the clock signal indicating at which time intervals a new CPU cycle starts, and then visualize how the processor executes the instruction stream over time. Clearly indicate the start and end of each of the 5 datapath stages (IF, ID, EX, MEM, WB) for all instructions. Note: you can assume the CPU starts from a clean state (e.g., after a system reset).*

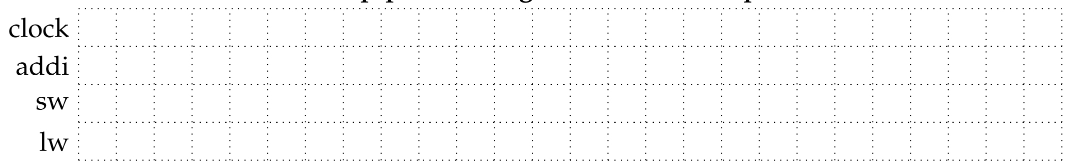
**single-cycle design (each cell is 100ps)**



**multi-cycle design (each cell is 100ps)**



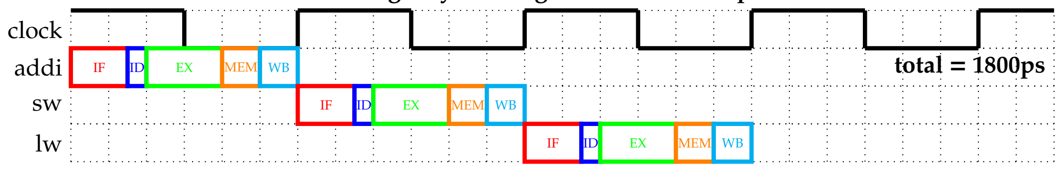
**pipeline design (each cell is 100ps)**



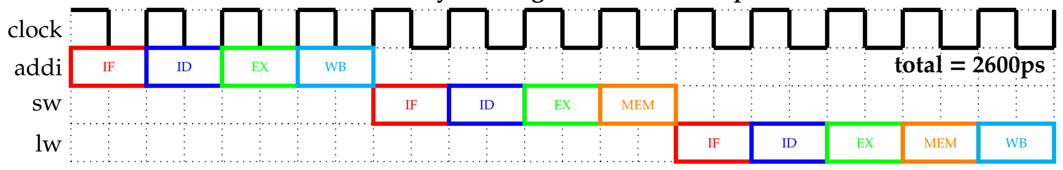
---

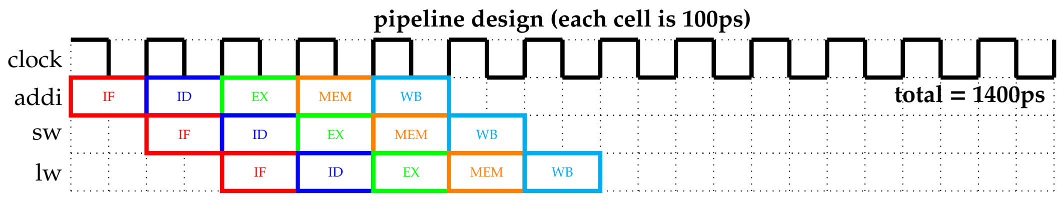
**Solution:**

**single-cycle design (each cell is 100ps)**



**multi-cycle design (each cell is 100ps)**





**Exercise 4.4.** *What is the total time and cycles needed to execute the above RISC-V program from question (4.3) for each of the three CPU designs? What if we add 50 no-operation instructions (*`add zero, zero, zero`*)? Provide a formula to explain your answer.*

**Solution:**

- single-cycle: 3 cycles (1800ps); 50 extra cycles (+30,000 ps)
- multi-cycle: 13 cycles (2600ps); 50*4 extra cycles (+40,000 ps)
- pipeline: 7 cycles (1400ps); 50*1 extra cycles (+10,000 ps)

**Exercise 4.5.** *Describe the performance implications for each of the three CPU designs, if in the above RISC-V program the first instruction is changed to* `addi t1, zero, 10`*. (hint: hazards)*

**Solution:** No change for single/multi-cycle, but this would create a pipeline data hazard that should be resolved in hardware with either forwarding logic from the EX to ID stages, or insertion of stall cycles (bubbles).

## 5 Forwarding

The code below describes a simple function in RISC-V assembly.

```
or t1,t2,t3
or t2,t1,t4
or t1,t1,t2
```

Assume the following cycle times for each of the options related to forwarding;

Without forwarding: 250ps

With Full Forwarding: 300ps

**Exercise 5.1.** *Indicate the dependencies and their type: Read After Write (RAW), Write After Read (WAR), Write after Write (WAW).*

**Solution:**

- RAW on t1 from I1 to I2 and I3
- RAW on t2 from I2 to I3
- WAR on t2 from I1 to I2
- WAR on t1 from I2 to I3

- WAW on t1 from I1 to I3

---

**Exercise 5.2.** *Assume there is no forwarding in the pipelined processor. Indicate hazards and add* `nop` *(no operation) instructions to eliminate them.*

---

**Solution:**

In the basic five stage pipeline WAR and WAW dependencies do not cause any hazards. Without forwarding, any RAW dependence between an instruction and the next two instructions (if register read happens in the second half of the clock cycle and the register write happens in the first half). The code that eliminates these hazards by inserting NOP instructions is:

```
or t1,t2,t3
nop
nop
or t2,t1,t4
nop
nop
or t1,t1,t2
```

Delay I2 to avoid RAW hazard on t1 from I1, Delay I3 to avoid RAW hazard on t2 from I2

---

**Exercise 5.3.** *Assume there is full forwarding in the pipelined processor. Indicate the remaining hazards and add* `nop` *(no operation) instructions to eliminate them. Compared the speedup achieved by adding full forwarding to a pipeline with no forwarding.*

---

**Solution:**

With full forwarding, an ALU instruction can forward a value to EX stage of the next instruction without a hazard. However, a load cannot forward to the EX stage of the next instruction (by can to the instruction after that). Th e code that eliminates these hazards by inserting NOP instructions is:

```
or t1,t2,t3
or t2,t1,t4 %No RAW hazard on R1 from I1 (forwarded)
or t1,t1,t2 %No RAW hazard on R2 from I2 (forwarded)
```

The total execution time is the clock cycle time times the number of cycles. Without any stalls, a three-instruction sequence executes in 7 cycles (5 to complete the fi rst instruction, then one per instruction). The execution without forwarding must add a stall for every NOP and overall, we get:

No forwarding: $(7 + 4) \times 250 = 2750$ ps

with forwarding : $7 \times 300 = 2100$ ps (1.23 speedup)

---

# 6 Code Optimization

The code below describes a simple function in RISC-V assembly(A = B + E; C = B + F;).

```
lw t1, 0(t0)
lw t2, 4(t0)
add t3, t1, t2
sw t3, 12(t0)
lw t4, 8(t0)
add t5, t1, t4
sw t5, 16(t0)
```

**Exercise 6.1.** *Assume the above program will be executed on a 5-stage pipelined processor with forwarding and hazard detection. How many clock cycles will it take to correctly run this RISC-V code?*

---

**Solution:**

Both add instruction will stall one cycle: 13 cycles (7 instructions + 2 stall cycles + 4 latency (5 pipeline stages))

---

**Exercise 6.2.** *Reorganize the code to optimize the performance. (Hint: try to remove the stalls)*

---

**Solution:**

```
lw t1, 0(t0)
lw t2, 4(t0)
lw t4, 8(t0)
add t3, t1, t2
sw t3, 12(t0)
add t5, t1, t4
sw t5, 16(t0)
```

11 cycles

---

## 7 Branching

The code below describes a simple function in RISC-V assembly.

```
        add x1, x0, x0
bar:
        bne x1, x0, exit
        bge x1, x0, foo
        addi x1, x1, -100
        add x5, x5, x5
        add x6, x1, x1
        sub x1, x1, x2
foo:
        addi x1, x1, 1
        jal  x10, bar
exit:
        xor x20,x21,x22
```

**Exercise 7.1.** *Fill out the following instruction/time diagram for the following set of instructions until the instruction on line 13 (`xor`) fully executes and commits. Execution starts from line 1.*

| PC | inst | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 | T16 | T17 | T18 | T19 | T20 | T21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x2000 | add | F | D | X | M | W | | | | | | | | | | | | | | | | |
| 0x2004 | bne | | F | D | X | M | W | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |

**Solution:**

| PC | inst | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 | T16 | T17 | T18 | T19 | T20 | T21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x2000 | add | F | D | X | M | W | | | | | | | | | | | | | | | | |
| 0x2004 | bne | | F | D | X | M | W | | | | | | | | | | | | | | | |
| 0x2008 | bge | | | F | D | X | M | W | | | | | | | | | | | | | | |
| 0x200C | addi | | | | F | D | - | - | - | | | | | | | | | | | | | |
| 0x2010 | add | | | | | F | - | - | - | - | | | | | | | | | | | | |
| 0x201C | addi | | | | | | F | D | X | M | W | | | | | | | | | | | |
| 0x2020 | jal | | | | | | | F | D | X | M | W | | | | | | | | | | |
| 0x2024 | xor | | | | | | | | F | - | - | - | - | | | | | | | | | |
| 0x2004 | bne | | | | | | | | | F | D | X | M | W | | | | | | | | |
| 0x2008 | bge | | | | | | | | | | F | D | - | - | - | | | | | | | |
| 0x200C | addi | | | | | | | | | | | F | - | - | - | - | | | | | | |
| 0x2024 | xor | | | | | | | | | | | | F | D | X | M | W | | | | | |