# Computer Architecture Exercises

## Series 5

Adrian Wälchli, Qiyang Hu

May 15, 2018

# Exam Information

| | |
|---|---|
| Date | Monday, June 4, 2018 |
| Time | 16:00 |
| Duration | 120 minutes |
| Location | ExWi A6 |
| Tools | 4 **handwritten** A4 pages (2 sheets of paper) |
| Bring | Legi |
| Registration | KSL |

# Static Branch Structure

```
        // s1 := 1024
        addi $s1, $zero, 1024
        // s1--
loop:   addi $s1, $s1,    -1
        // call subroutine()
        jal  subroutine
        // if (s1 != 0) jump loop
        bne  $s1, $zero, loop
```

- prediction is wrong 1024 times
- "Predict not taken" does not work well with "bottom of the loop"
- better: "top of the loop"
- see slide 17 in "Containing Control Hazards"

# Static Branch Structure

Optimized code for “Predict not taken”:

```
        // s1 := 1024
        addi $s1, $zero, 1024
        // s1--
loop:   addi $s1, $s1, -1
        // if (s1 == 0) stop loop
        beq  $s1, $zero, out
        // call subroutine()
        jal  subroutine
        // continue loop
        j    loop
out:
```

- only last prediction is wrong

# Antidependencies

- dependencies between two instructions in the context of Out-of-Order Issues
- subsequent instruction overwrites a value an earlier instruction did not used/read yet.
- write before read
- compare to data hazard: subsequent instruction reads a value that has not yet been produced by earlier instruction (read before write)
- Example:

  R3:= R3 * R5;
  R3:= R3 + 1;

  If result of addition is written before completion of multiplication, result of multiplication is wrong
- see slide 16 in "Multiple Issue Introduction"

# Branch Delay Slot

- at branch instruction, next instruction can not be determined immediately
- need to wait until branch condition is evaluated (stall)
- or: instead of stalling, processor can execute instruction that have no dependencies (that need to be executed anyway)
- Example: Decrement counter value in a loop

```
        // do something...
loop:   jal   subRoutine
        // repeat
        bne   $t1, $zero, loop
        // branch delay slot
        addi  $t1, $t1, -1
```

# Dynamic Branch Structure

```
addi $s1, $zero, 0          // i := 0
outerLoop:

addi $s2, $zero, 0          // j := 0
innerLoop:
        // t = t + i * j
  addi $s2, $s2, 1          // j++
  bne  $s2, 10, innerLoop   // if (j<10) repeat

addi $s1, $s1, 1            // i++
bne $s1, 10, outerloop      // if (i<10) repeat
```

- Assumption: predict "taken" (1/10) at beginning
- 1-bit predictor: $2 \cdot 9 + 1 = 19$ wrong predictions
- 2-bit predictor: $9 \cdot 1 + 2 = 11$ wrong predictions
- see slide 22 in "Containing Control Hazards"

# Multiple-Issue Processors

## True or false?

Pipelining takes advantage of instruction-level parallelism, but does not enable the processor to execute more than one instruction at a time.

## Answer

**False.** Most instructions can not be performed in one clock but require multiple stages of processing. Therefore we can split an instruction into multiple stages and keep every stage "busy" by executing different instructions at the same time.

# Multiple-Issue Processors

## True or false?

Multiple-issue processors are able to execute more than one instruction per clock cycle.

## Answer

**True.** This requires extending the traditional pipeline by repeating fundamental blocks like the ALU.
Two types: *static* and *dynamic*.
Need additional logic to avoid structural hazards.

# Multiple-Issue Processors

## True or false?

SIMD (single-instruction multiple-data) processors are static multiple-issue processors.

## Answer

**True.** Data-parallelism is scheduled at compile time.
Example: operations on arrays, matrix multiplication, etc.