

# CSC2/452 Computer Organization

## The Processor Pipeline

Sreepathi Pai

URCS

October 16, 2019

# Outline

Recap

Instruction Execution

Instruction Pipelining

Software and the Pipeline

# Outline

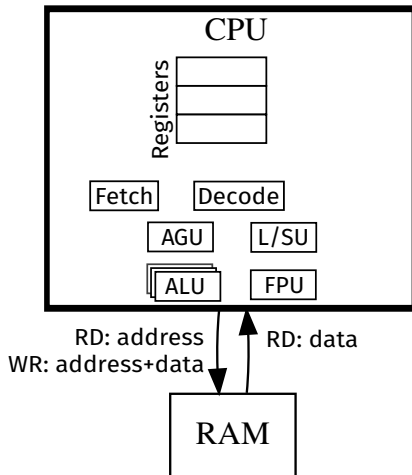
Recap

Instruction Execution

Instruction Pipelining

Software and the Pipeline

# The Processor and RAM (so far)



# Recall hellopi execution statistics (lecture 1)

```
$ perf stat -e instructions ./a.out  
Hello, the value of pi is 3.141593
```

```
Performance counter stats for './a.out':
```

```
        662,172      instructions
```

```
0.001168841 seconds time elapsed
```

# The Problem

How do you execute billions of instructions as quickly as possible?

# The Performance Equation

$$T = \frac{W \times t}{P}$$

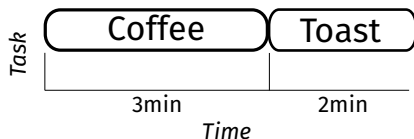
- ▶  $W$ , work items to be completed
  - ▶ e.g., instructions to be executed
- ▶  $t$ , the average time per work item
  - ▶ cost per work item
- ▶  $P$ , average parallelism
  - ▶ Number of work items that can be executed in parallel
- ▶  $T$ , total time for execution

# Example: Preparing Breakfast

- ▶ Work item 1: Toast
  - ▶ Time for making toast: 2 minutes
- ▶ Work item 2: Coffee
  - ▶ Time for making coffee: 3 minutes
  - ▶ Not instant!



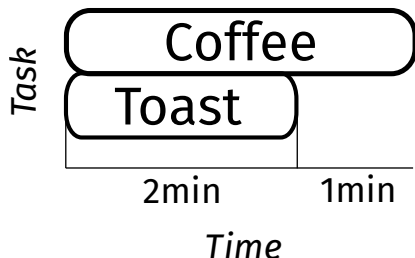
## Prep. Time for Breakfast: Serial



- ▶ Toast: 2 minutes, Coffee: 3 minutes
- ▶ Only Stove available to make breakfast
  - ▶ Average Parallelism:  $P = 1$
  - ▶ Work items  $W = 2$
  - ▶ Average Time  $t = (3 + 2)/2$
  - ▶ Total Time: ?

## Prep. Time for Breakfast: Parallel

- ▶ Toast: 2 minutes, Coffee: 3 minutes
- ▶ Stove *and* Toaster available
  - ▶ Total Time: 3 minutes
  - ▶ Work items  $W = 2$
  - ▶ Average Time  
 $t = (3 + 2)/2$
  - ▶ Average Parallelism:  $P = (2 \times 2 + 1 \times 1)/3 = 1.66$
- ▶ Speedup (serial time/parallel time) is  $5/3 = 1.66$ 
  - ▶ But, requires more equipment



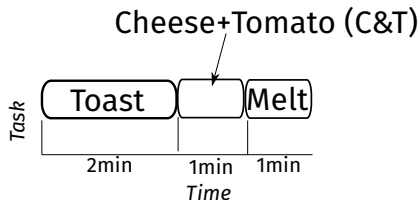
# Other ways to speed up breakfast

- ▶ Don't eat toast
  - ▶ Decreases work,  $W$
- ▶ Drink instant coffee
  - ▶ Decreases average time,  $t$

# Speeding up Programs

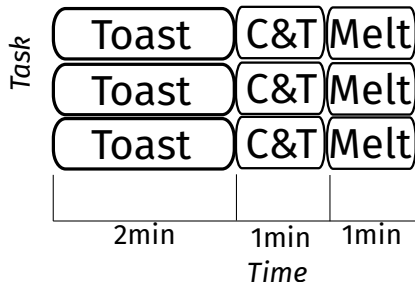
- ▶ Do less work (i.e., decrease  $W$ )
  - ▶ fewer instructions
  - ▶ choose algorithms with fewest operations
- ▶ Do cheaper work (i.e., decrease  $t$ )
  - ▶ not all instructions have the same cost
  - ▶ e.g. integer multiplies are slower than integer shifts
  - ▶ the algorithm with lower constant costs is better
  - ▶ purpose of this course: to teach you which instructions are cheap and which are expensive
  - ▶ Moore's Law gave us free decreases in  $t$
- ▶ Increase parallelism (i.e. increase  $P$ )
  - ▶ only option left if you've already reduced  $W$  and  $t$
  - ▶ take CSC258 to know more

## Example: Making a Sandwich



- ▶ Toast 2 bread slices (2 minute)
- ▶ Add cheese and tomato slices (1 minute)
  - ▶ Assuming tomatoes and cheese have to be sliced
- ▶ Melt cheese (1 minute)
- ▶ Total time: ?
- ▶ Average Parallelism: ?

## Making 3 sandwiches

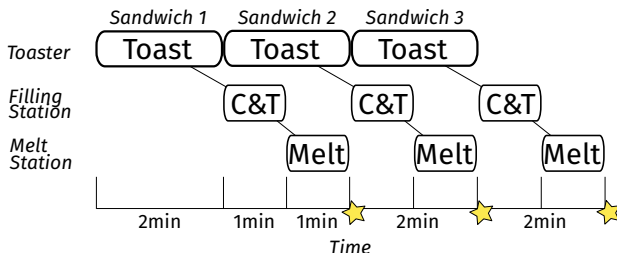


- ▶ Total time: ?
- ▶ Average Parallelism: ?
- ▶ Equipment needed: ?
  - ▶ Toasters?
  - ▶ Knives?
  - ▶ Stove? [assume one stove can melt cheese in one sandwich]
  - ▶ Workers?

# Making sandwiches: Observations

- ▶ Making one sandwich takes 4 minutes, end-to-end
  - ▶ Each step is dependent on previous step, no parallelism
- ▶ Making multiple sandwiches is highly parallel
  - ▶ Making each sandwich is independent of the other
  - ▶ But exploiting that parallelism requires lots of equipment
  - ▶ Also, as far as sandwich eaters are concerned, they get a one sandwich every 4 minutes

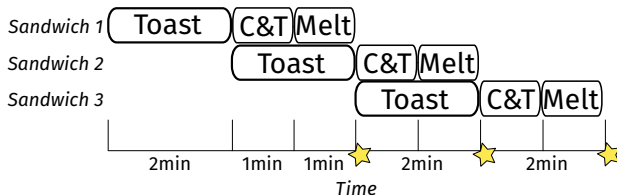
# Pipeline Parallelism in the Kitchen



- ▶ Equipment needed?
  - ▶ Toasters (Toaster station): ?
  - ▶ Knives (Filling station): ?
  - ▶ Stove (Melt station): ?
  - ▶ Workers?
- ▶ Time for first sandwich?
- ▶ Time for subsequent sandwiches (indicated by stars)?
  - ▶ Each sandwich still takes 4 minutes to make



## Alternate View: Pipeline Parallelism in the Kitchen



- ▶ Same timeline as previous figure, but from perspective of sandwich instead of a work station

## Some performance measures

- ▶ Sandwich Latency (also Job Service Time): 4 minutes
  - ▶ Also pipeline fill time
- ▶ *Service time* (or System Latency) for making sandwiches: 2 minutes
  - ▶ Time after pipeline is full
- ▶ *Throughput* (rate) of making sandwiches: 1 sandwich every 2 minutes

# Pipeline Parallelism

- ▶ Used to parallelize many similar (but independent) tasks
- ▶ Each task consists of highly dependent steps
- ▶ Pipelines consist of (sequential/serial) stages
  - ▶ Usually correspond to steps of each task
- ▶ But each stage can be handling steps from different tasks at a given time
  - ▶ e.g. at Time 5: sandwich 3 in Toaster and sandwich 2 in Filling
- ▶ Does not require more equipment than that required by one task
  - ▶ May require more “workers”

# You're not in culinary class!

How do these techniques apply to instruction execution (our original problem)?

# Outline

Recap

**Instruction Execution**

Instruction Pipelining

Software and the Pipeline

# Steps in Instruction Execution

Instruction execution in a processor consists of at least 3 steps:

- ▶ Fetch instruction
- ▶ Decode instruction
- ▶ Execute instruction

Each step is dependent on the previous one. But there are many instructions in a program!

# Program Execution: Example #1

```
I1: movq -8(%rbp), %rax
```

```
I2: addq %rcx, %rdx
```

Are I1 and I2 independent of each other?

- ▶ I1:
  - ▶ reads memory: `-8(%rbp)`
  - ▶ writes register: `%rax`
- ▶ I2:
  - ▶ reads register: `%rcx`
  - ▶ writes register: `%rdx`

## Program Execution: Example #2

```
I3: movq -8(%rbp), %rax
```

```
I4: addq %rax, %rdx
```

Are I3 and I4 independent of each other?

- ▶ I3:
  - ▶ reads memory: `-8(%rbp)`
  - ▶ writes register: `%rax`
- ▶ I4:
  - ▶ reads register: `%rax`
  - ▶ writes register: `%rdx`



# Data Dependences

- ▶ I1 and I2 are independent
  - ▶ Reads and writes do not overlap
- ▶ I3 and I4 are dependent
  - ▶ I3 produces (i.e. writes) value that is consumed (i.e. read) by I4
  - ▶ I4 execution must wait until I3 produces a value

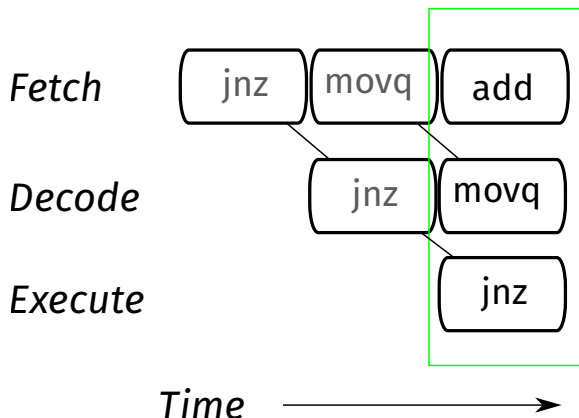
## Program Execution: Example #3

```
test %eax, %eax
jnz L1
movq ...
add ...
div ...
jmp ...
```

```
L1:
xor %eax, %eax
ret
```

- ▶ Which instruction follows `jnz`?
- ▶ Which instruction follows `ret`?

## Pipeline State



- ▶ If jnz does not jump, then current state of pipeline (green box) is valid.
- ▶ If jnz does make the jump, what must we do?
  - ▶ can't execute movq and add

# Control Dependences

- ▶ Can't decide which instruction follows `jnz` until it finishes execution
  - ▶ Could be `movq` if `%eax` is zero
  - ▶ Could be `xor` (at L1)
- ▶ Can't decide which instruction follows `ret` either
  - ▶ Need to look at function stack to find return address
- ▶ In these cases, the instructions we have fetched and decoded may be wrong
  - ▶ Usually, we need to throw them away
  - ▶ Called a "pipeline flush"

## Program Execution: Example #4

```
divl %ecx  
addq %edi, %esi
```

- ▶ Both `divl` and `addq` use the ALU
- ▶ But `divl` takes more time than `addq`
  - ▶ `addq` must wait until `divl` is done

# Structural Dependence/‘Hazards’

- ▶ Different instructions may require use of the same (type) of resources
- ▶ Instructions must wait until “structure” (e.g. functional unit) is free
  - ▶ Could be other structures, e.g. queues
- ▶ Kitchen example:
  - ▶ Toast + Coffee with only one stove and no toaster
  - ▶ Can only toast or make coffee at the same time, not both

# Instruction Execution: The Problem

- ▶ Could be pipelined
  - ▶ Each instruction consists of multiple steps
  - ▶ There are many instructions that must be executed
- ▶ Complication: Not all instructions are independent
  - ▶ Data dependence: Instruction requires results from previous instruction
  - ▶ Control dependence: Next instruction depends on execution of current instruction
  - ▶ Structural dependence: Instructions may require use of same structure or some structure is full
- ▶ Other complications
  - ▶ Instructions can take different execution times (e.g. `divl` vs `addq`)

# Outline

Recap

Instruction Execution

**Instruction Pipelining**

Software and the Pipeline



# Stages

Although a 3-stage pipeline can be used (and sometimes fewer stages can also be used, usually by eliminating decode<sup>1</sup>), usually 5 stages are used:

- ▶ Instruction Fetch (IF)
- ▶ Instruction Decode (ID)
- ▶ Instruction Execute (EX)
- ▶ Memory (MEM)
- ▶ Instruction Writeback (or retirement) (WB)

The MIPS R2000 (the first MIPS processor from Stanford) popularized the 5-stage pipeline.

---

<sup>1</sup>See the Berkeley RISC I processor

# EX/MEM

```
addq %rax, %rdx          # %rdx = %rdx + %rax
movq (%rdi, %rsi, 2), %rbx # %rbx = memory[%rdi + %rsi * 2]
```

- ▶ Note how `movq` is computing an effective address and then loading data from memory
  - ▶ Two steps combined into one
- ▶ EX stage handles all arithmetic computation
- ▶ MEM stage is used to access memory
  - ▶ skipped if the instruction does not access memory

- ▶ The writeback stage isn't always necessary in simple designs
  - ▶ But if you can flush a pipeline, then you need to prevent wrong instruction results from being made visible
- ▶ Stage where results are made permanent (sometimes called "Commit")
  - ▶ Values written back to registers
- ▶ Also releases any resources occupied by instruction

# Dealing with dependences

- ▶ General solution, add an interlock
- ▶ This is a circuit that delays instructions
  - ▶ introduces “bubbles” (a no-op) by stalling a stage (stops the stage) in the pipeline

## More sophisticated solutions: Use a compiler!

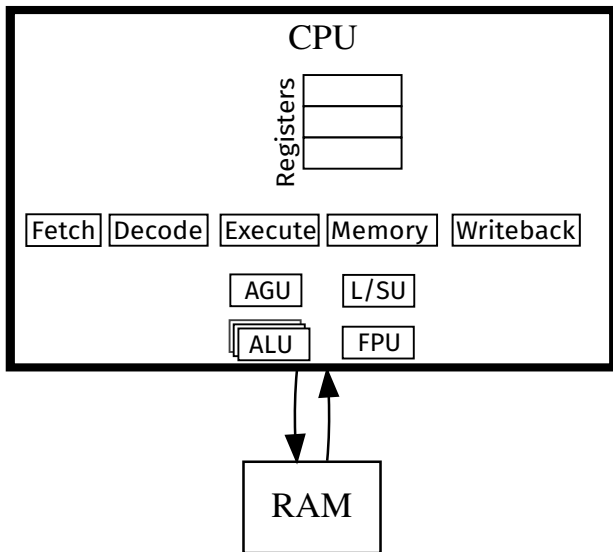
- ▶ Reorder instructions to:
  - ▶ keep dependent instructions far from each other
  - ▶ prevent structural hazards
  - ▶ reordering must preserve program semantics!
- ▶ Use branch-delay slots for control dependence
  - ▶ Instruction after branch is always executed, regardless of direction of branch
  - ▶ Can be a NOP instruction
  - ▶ Now considered to be a poor design practice
- ▶ These techniques were used by MIPS (Microprocessor without Interlocked Pipelined Stages)

# Terminology

- ▶ Superscalar processor: A processor that can execute more than one instruction at the same time
- ▶ In-order pipeline: Instructions are executed in (assembly-language) program order
  - ▶ Note: like on MIPS, the compiler may reorder instructions, but the pipeline doesn't
- ▶ Out-of-order pipelines: Instructions are executed out-of-order
  - ▶ In-order fetch
  - ▶ Out-of-order execute
  - ▶ In-order retirement
  - ▶ Most modern high-performance processors
- ▶ Speculative pipelines: Pipelines that guess which direction a branch is going to execute
  - ▶ Check guess at writeback stage and either flush or commit

This course looks at in-order pipelines, for the others, take CSC251.

## A view of the processor



# Outline

Recap

Instruction Execution

Instruction Pipelining

Software and the Pipeline



# Does the pipeline affect how you write software?

- ▶ Knowledge of the pipeline is crucial to understanding performance issues
- ▶ You can get information on pipeline behaviour using the `perf` tool on Linux
  - ▶ we'll see next class

# References

- ▶ A detailed description of the pipeline is given in Chapter 4 of the textbook
- ▶ We will be more software-centric than hardware-centric in this course
  - ▶ Take CSC251 if you want to really build a processor at the hardware level