L1: Introduction

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

September 11th, 2020

Architecture Research Group

SSE, TGGS

Administrative Stuff

Class Website

- Please sign-up on Canvas
  - Sign-up link: **https://canvas.instructure.com/enroll/JK8R6L**

- This is where all the information from this class is posted
  - Class policy and syllabus
  - Class schedule
  - Announcement
  - Assignments

Class Policy

- **No plagiarism**
  - Everything will have to be from your own work
  - You need to put proper citations/references to your source
    - Max(grade) * number of times you got caught
- **5 late days total, 2 per assignment max**
- **Office hours:** on Discord, TBA
  - I will likely have two slots a week
- I encourage you to discuss material with your classmates and work together, **but each student must**
  - <u>Write his/her own code</u>
  - <u>Clearly indicate who you have worked with</u>

Grading Breakdowns

- Assignments 30%
- Project 20%
- In-class exercise 10%
- Quiz 20%
- Final 20%

- I can curve anything above to make sure everything is fair

Class Project

- Open-end
  - Build whatever you want, but they should utilize knowledge you learn from this class
- We will kick start this after the midterm
  - But you are all welcome to discuss your ideas as early as right after this lecture
- Some potential ideas:
  - Write a parallel version of known algorithms
  - Try out CUDA

Language Used in This Class

- We will use a few languages to show different concepts
  - Scala
  - Rust
  - C++ (for OpenMP)

Text

In-class Exercise

- There will be both lecture slides and coding exercises

- Lecture will be at most 3 hours
  - Usually will be around 1.5 – 2.5 hours
  - There will be a longer break in the middle
    - Feels free to eat during class

- Then, you will do an in-class exercise

My Expectation

- There will be a lot of new way of coding
  - Functional programming will feel very different than imperative programming
  - Applies to both the assignments and the project
- Workload will be heavy
  - Start your assignment early is always a good idea
- You should have a good grasp of
  - Intro to programming (Python)
  - Intermediate programming (JAVA)
- You should have some basic on
  - Computer system
  - Computer hardware

What Will You Learn?

The Goal of This Course

- You should be able to:
  - Know essential concepts related to programming languages
  - Know the benefit of parallel programming
  - Know how to increase parallelism (more performance)

Designing a Programming Languages

Design Tradeoffs for Prog. Lang.

- Syntax and complexity of the code → Grammar in English lang.
- Semantics → what does the paragraph mean.
- Paradigms that the language favors
- Type system and type rules
- Memory management
- Need a compiler?

Programming Languages Over Time

- Early day (1950s – 1960s)
  - Language mirrors hardware concepts
    - Compiler optimization is expensive and mostly impossible
  - Programmer is much cheaper compare to machines
    - Parts are costly
    - Programs has to be very efficient from the get-go
- Now
  - Language centers on design concepts
    - Includes things like objects, records, functions
  - Machine is cheap and will continue to be cheaper
    - Scripting and inefficient codes are(???) ok, quick to develop
  - Optimized for resource constraints and design goals
    - Low power
    - High throughput, high parallelism

Why So Many Languages?

- Have you notice there are many languages?

- Have you notice each one of them offer different tradeoff?
  - Ease-of-use
  - Safety
  - Performance
  - Etc.

Emergence of Parallelism

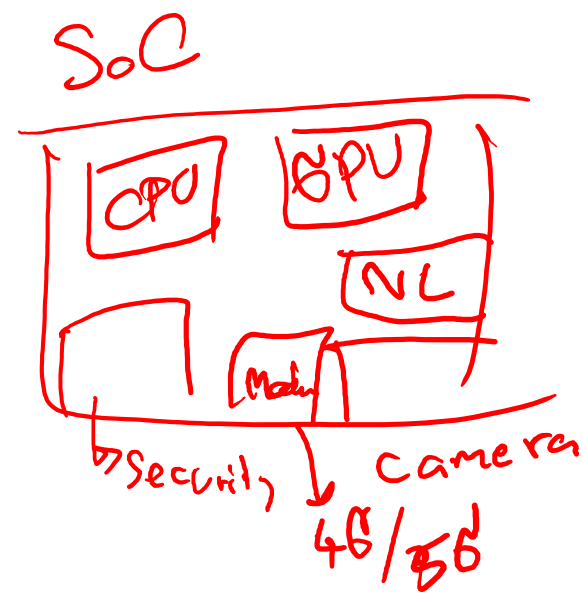
von Neumann Model (Common)

- Stored-program computer
- Two key properties
  - Programs (instructions) are stored in a linear memory array
  - Memory is unified between instructions and data
    - Control signal interpret whether stored values are data or instructions
- Sequential instruction processing
  - One instruction at a time
    - Fetch → executed → complete
  - Program counter (PC) identify the current instruction
    - PC is also referred to as Instruction Pointer (IP)
  - Program counter advanced sequentially except for control transfer instruction (e.g., branches)

program

The von Neumann Model

- Is this the only model? No
- But this is one of the most dominant
- All major instruction set architectures (ISA) today use this model
    - x86, ARM, MIPS, SPARC, Alpha, POWER



- What is the alternative?

The Dataflow Model

$f(n) = \ldots$

- Von Neuman: An instruction is fetched and executed in **control flow order**
  - Instruction pointer grabs the next instruction
  - Mostly sequential except control flow instructions

control flow

$g(n)$

$f(m)$

$h(p)$

- Dataflow model: An instruction is fetched in the **data flow order**
  - Compute when operands are ready
  - No instruction pointer
  - Ordering is based on data flow dependence
    - Think of a math function
  - Many instruction can execute at the same time
    - **Parallelism** ☺

von Neumann vs. Data Flow

Sequential

1 sec C = A + B;

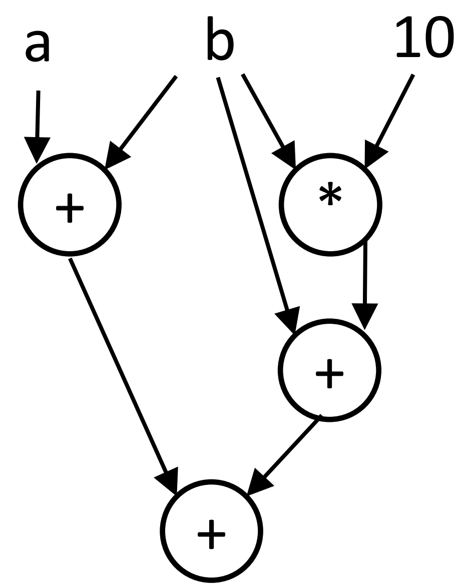
1 sec X = B * 10;

1 sec Y = B + X;

1 sec Z = C + Y;

4 secs

Dataflow



1 sec

1 sec

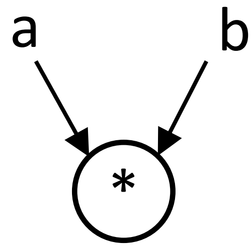
1 sec

3 sec

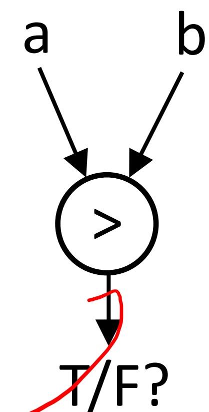
- Which is more natural as a programmer?

Types of Dataflow Nodes

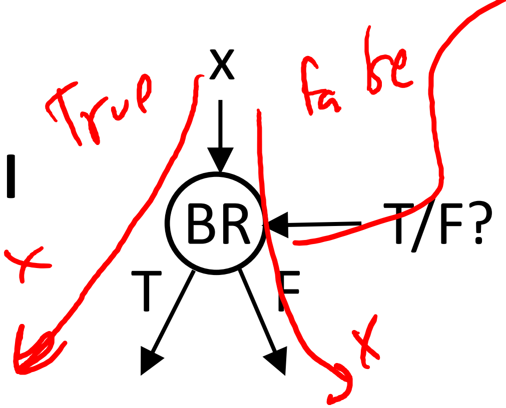
Computation



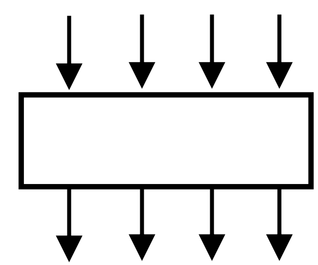
Relational



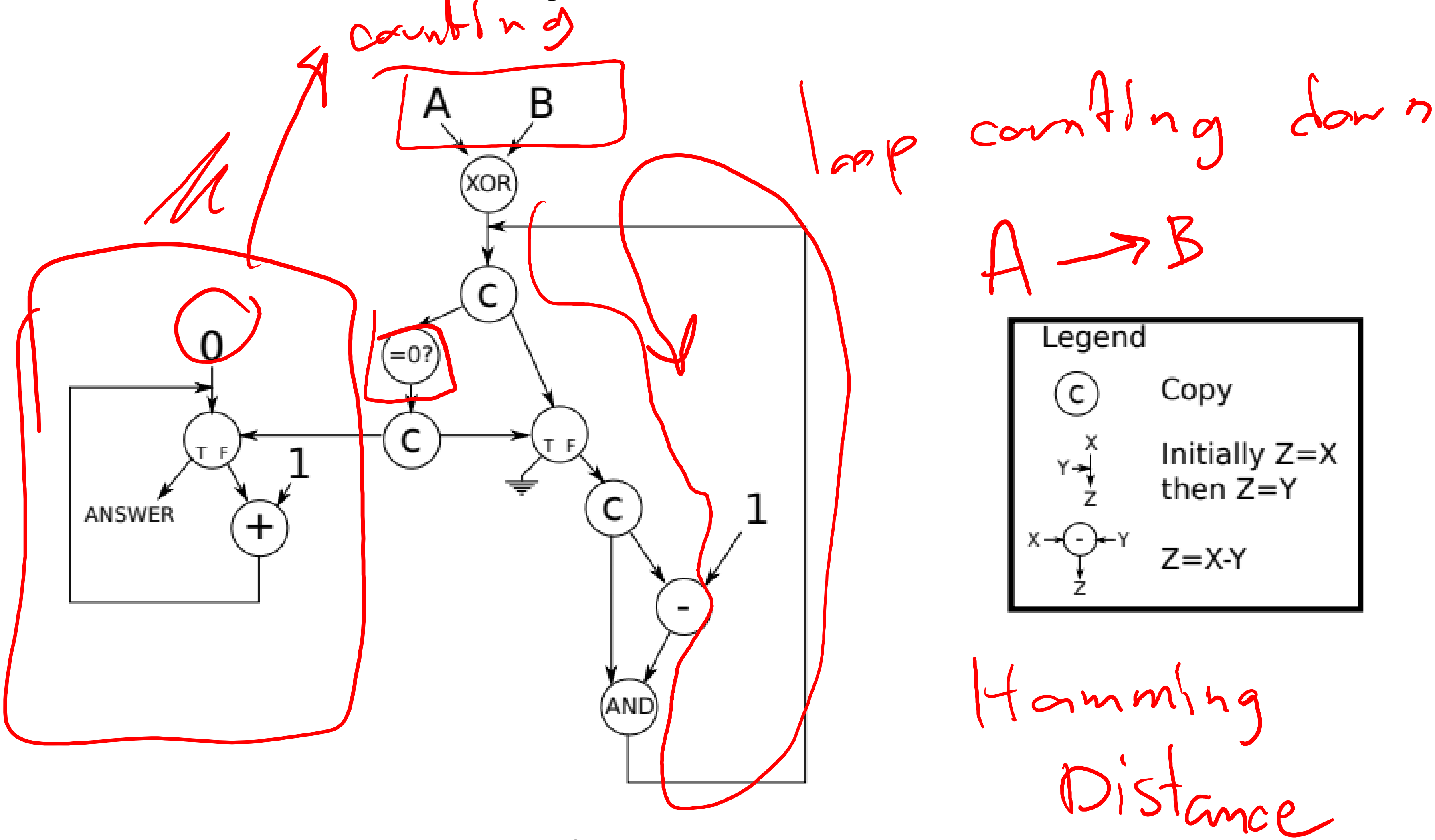
Conditional



Barrier/Synch



In-class Group Exercise



- What does this dataflow program do?
  - Hint: do one side at a time

Let’s Dive into Func. Programming

Expressions

a + b

- What are expressions?
  - 10
    - Expression that evaluate to 10, has type Int
  - 12 + 13 → 25
    - Expression that evaluate to 25, type Int

name

- You can bind a name to expression
  - def number = 10 → expr
    - This gives number: Int
- You can combine expressions
  - number * 10
- Expression does not always have a type
  - 3*”Hello”

Expressions Definition

- This is called named expression
- You can think of this as a math function

- Example:
  - def cube(x: Double) = x*x*x
  - def ssc(x: Double, y: Double) = cube(x) + cube(y)

Substitution Model

- When evaluating an expression, you can use substitution
- Example: Assume def f(n:Int) = n*n
  - We want to evaluate f(2+1)

- First, 2+1 is evaluated to 3
  - Then, every time we see ~~f~~ n as its expression
    - We replace it with 3
- f(2+1) → f(3)
  → {n*n} [n← 3]
  → 3*3
  → 9

Termination

- If everything is a function, when does the evaluation of an expression reduces to a value
- Question:
  - Does every expression reduce to a value in finite step?
- Let’s look at this seemingly confused example:
  - def loop: Int = **loop**
- **loop** has a type Int, but <u>never terminate</u>
- Our substitution model replaces loop with loop …
  - And this goes on indefinitely
- So, not every expression reduce to a value in finite step

Another Evaluation Strategy

- So far, we use the substitution model to evaluate exp.
- Let’s experiment with a different strategy:
  - Idea: Pass the arguments into the function w/o reducing them

f(2+1) → {n*n} [n← 2+1]
→ (2+1)*(2+1)
→ 9

- This evaluation strategy yields the same result
- Why? Because our computation has no side effect!
  - I.e., the order of substitute vs. reduce does not affect the final result

Different Function-calling Style

- Call by value (CBV)
  - Reduce first, then substitute
- Call by name (CBN)
  - Substitute first, then reduce

- Both strategies should evaluate to the same final value

Theorem on CBV/CBN

- Both strategies reduce to the same final values as long as
  - All expressions involved are pure functions (i.e., no side effect)
  - Both evaluation terminates
- Furthermore:
  - If CBV of expression e terminates, then CBN of e terminates
  - Does not true for the other direction!

- CBV → every function's argument is evaluated once
- CBN → no evaluation if unused in the function body
- Scala is CBV by default
  - You can invoke CBN by annotating input param with the type
    - Def addTwo(x: => Int) = x+2

Let’s Play Around

- Consider
  - def leftCBV(x:Int, ~~y:Int~~) = x
  - def leftCBN(x:=> Int, y: ~~=>Int~~) = x
  - def loop:int = loop
- Try to call the two version with
  - 1 leftCBV(1+1,loop) and 2 leftCBV(loop, 1+1)
  - leftCBN(1+1,loop) and leftCBN(loop, 1+1)

    3 4

  - What happen?

Conditional Expressions

- Scala offers the if-then-else construct
  - It tell which ***expression*** to step to next
    - Vs. which statement/commands to proceed with

- Example
  - def abs(x:Int) =
    if (x<=0) –x else x

- Using the construct, we can say if (e1) e2 else e3 behave:
  - e1 => true [if(e1) e2 else e3] → e2
  - e1 => false [if(e1) e2 else e3] → e3

} either e2 or e3

Example

- Let’s evaluate abs(-40)

→ [if(x<=0) –x else x]/[x = - 40]
→ if(-40 <=0) – (-40) else - 40
→ if(true) –(-40)else -40
→ - (-40)
→ 40

- Let’s try abs(5)

→ [if(x<=0) –x else x]/[x=5]
→ if(5 <=0) – (5) else 5
→ if(false) –(5)else 5
→ 5

More Complex Example

- def loop: Int = loop
- def goof(x:Int) = if (x<0) loop else 10

- What happen if we run goof (1) vs. good (-1)

f

e1 e2 e3

(1<0) loop else 10

(false) ~~loop~~ else 10

10

loop

Reduction on Boolean Expression

- Takes two basic values: True and False
- Evaluating the expression following normal logic op.

e : Boolean

!true → false

true && e → e

true || e → true

!false → true

false && e → false

false || e → e

What Does “def” Do?

- def binds and expression to a name

- So, fundamentally, def is a “by-name” type
  - The right-hand expression is not evaluated until used

- If we want to use a by-value form, use “val”

  def foo1 = 11
  val foo2 = 11

Example

- Suppose x:Boolean and y:Boolean
- We want to simulate && and ||
  - Remember that **they are short circuit**: false && loop = false

- Answer:
  - def and(x: Boolean, y => Boolean) = if(x) y else false
  - def or(x: Boolean, y => Boolean) = if(x) true else y

Nested Functions

- Example
  - ```
    def sumOfSquares(x:Int, y:Int) = {
      def sqr(t:Int) = t*t
      sqr(x) + sqr(y)
    }
    ```

return value

sqr doesn't exist

- This helps namespace pollution
  - sqr only seen inside sumOfSquare
  - Also notice the last statement of {…} is the return value
    - I.e., it determine what sumOfSquare evaluates to
    ```

Blocks

- The following is valid
```
{
    val number = 10
    number+1
}
```
- Extendind this idea, we can do
```
def foo = {
    val number = 10
    number+1
}
```
- This binds foo to the expression inside the brace

~~{ val num = 10~~
~~}~~
~~num+1~~

foo

Visibility

- Definition inside a block is only visible inside
- Definition inside shadows things defined outside the block

- Example: What is the outcome of

val x=5 x_1
val result = {
        val x = 6 x_2
        x+1
}
println(x) $x_1 \rightarrow 5$
println(result) $x_2+1 \rightarrow 7$

Before We Leave Today

Make Sure You Have Scala

- Please install it right now
  - https://www.scala-lang.org/download/

- Try to run this following code:

```
object HelloWorld extends App {
println("Hello, World!")
}
```

- The code should print Hello, World!

Scala REPL

- REPL
  - **R**epeat
  - **E**valuate
  - **P**rint
  - **L**oop

- Expression can be entered directly into the REPL