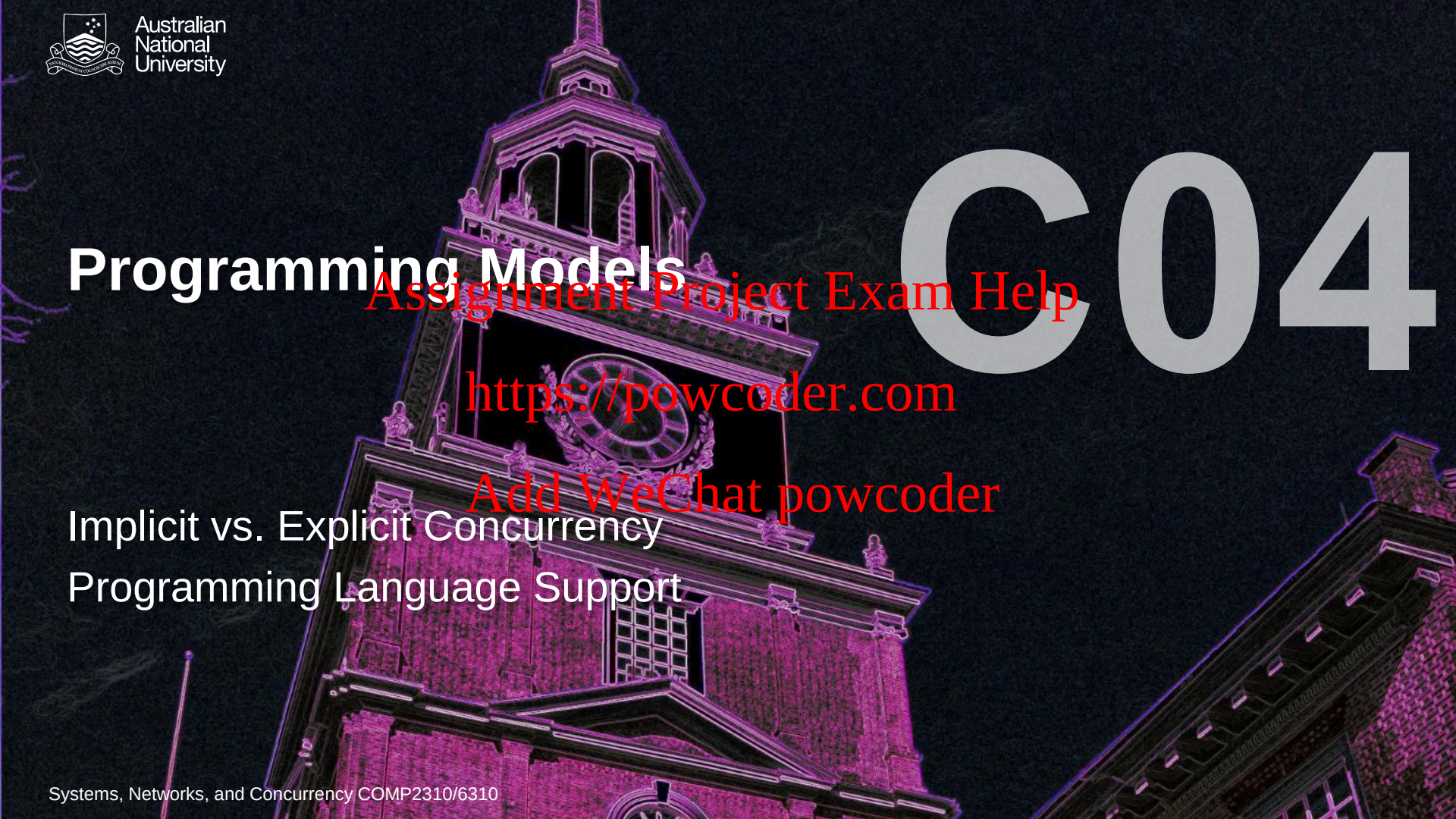

Australian National University

C04

Programming Models

Implicit vs. Explicit Concurrency

Programming Language Support

Programming Models for Concurrency

- Required: representation of concurrent entities (processes, threads, …)
- Desired:
  - management of concurrent entities (create, terminate, …)
  - resource protection (memory, devices, …)
  - contention management (mutual exclusion, …)
  - synchronization (semaphores, monitors, …)
  - communication (message passing, remote procedure calls, shared memory, …)

Explicit Concurrency

- Ada, C++, Rust, ~~Ruby, Python~~
- C# , other .net languages
- Java, Scala
- CSP, occam
- Erlang
- Go
- Swift
- Chapel, X10, UPC++, Charm++
- Algol 68, Modula-2, Modula-3

Implicit Concurrency

- functional languages: Lisp, Haskell, (O)Caml, Miranda, Spark …
- Prolog
- dataflow: Esterel, Lustre, Signal, TensorFlow, Legion, PaRSEC, …

Implicit Concurrency in Functional Languages

- Quicksort in a functional language (here: Haskell):

```
qsort[] = []
qsort(x:xs) = qsort[y | y <- xs, y < x] ++ [x] ++ qsort[y | y <- xs, y >= x]
```


- Pure functional programming is side-effect free
  - Parameters can be evaluated independently; could run concurrently

- Lazy evaluation:

```
borderline = (n /= 0) && (g(n) > h(n))
```


  - If n = 0 then the evaluation of `g(n)` and `h(n)` can be avoided
  - Concurrent evaluation should be interruptible
- Short-circuit evaluation assumes explicit sequential execution:

```
if Pointer /= nil and then Pointer.next = nil then ...
```

Library Support

- message passing: MPI
- POSIX processes
- POSIX threads (shared memory)