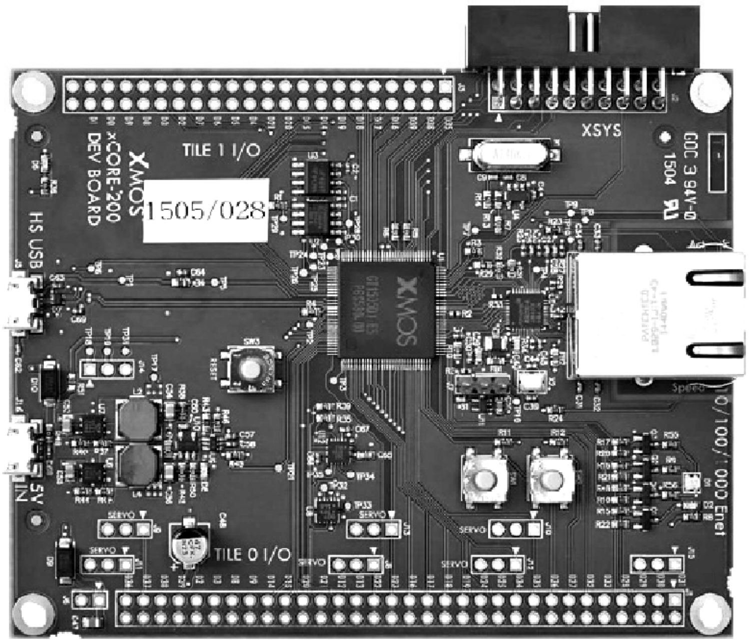
Department of Computer Science
University of Bristol

COMS20001 - Concurrent Computing

www.ole.bris.ac.uk/bbcswebdav/courses/COMS20001_2018/content



Lecture 02

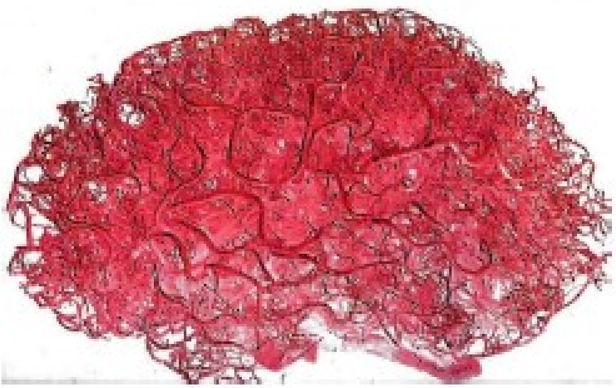
Towards Concurrent Programming in xC

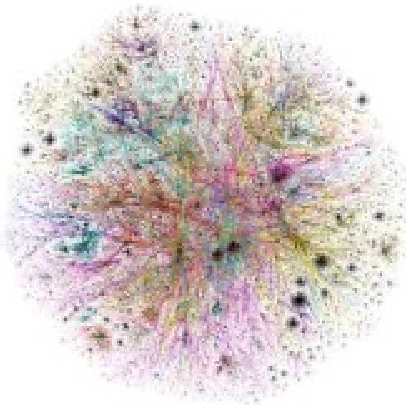
Sion Hannuna | hannuna@cs.bris.ac.uk
Tilo Burghardt | tilo@cs.bris.ac.uk
Dan Page | daniel.page@bristol.ac.uk

20 Slides

Recap: The Natural World is NOT serial ☺

- …**NATURE is massively concurrent** !
  - natural networks tend to be continuously evolving, yet they are robust, efficient and long-lived
  - Concurrency is one of nature's core design mechanisms – and one of ours!



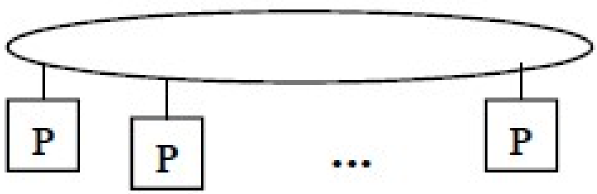


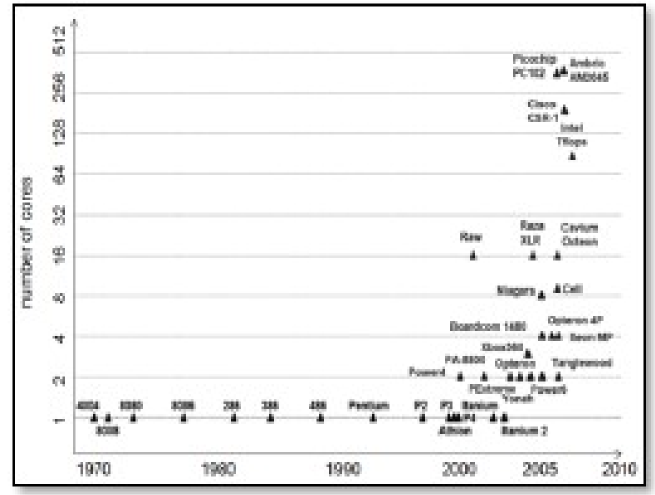
- in many cases **computing models phenomena of the real world**

  → computers are built as part of the physical world and can harvest natural concurrency for their own performance

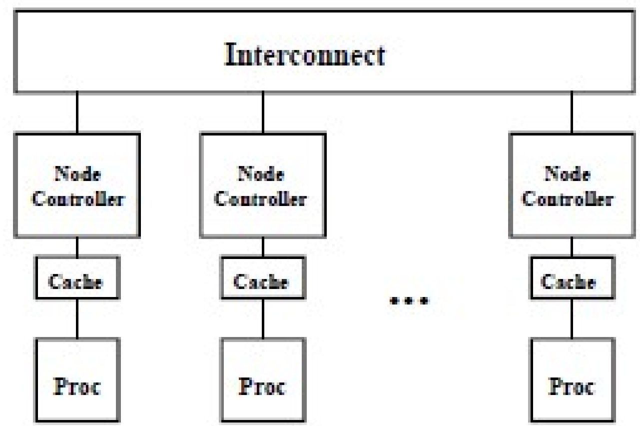
  → concurrency can often help simplifying the modelling of systems

- **Multiprocessors**
  (collection of communicating processors)
  - speed advantage by physically parallelised computation





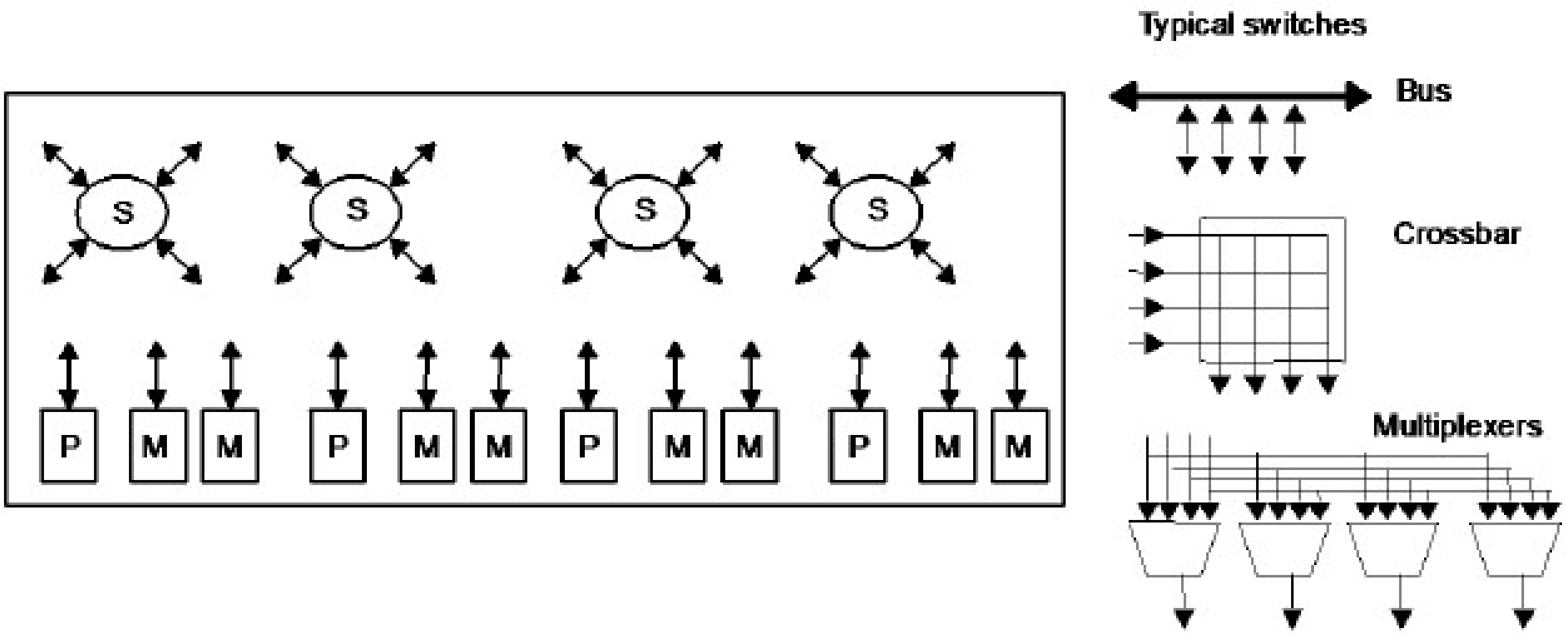
- **Multi-Memory Systems**
  - local, CPU-associated memory essential regardless of programming model
  - however, connectivity model affects specific performance tradeoffs



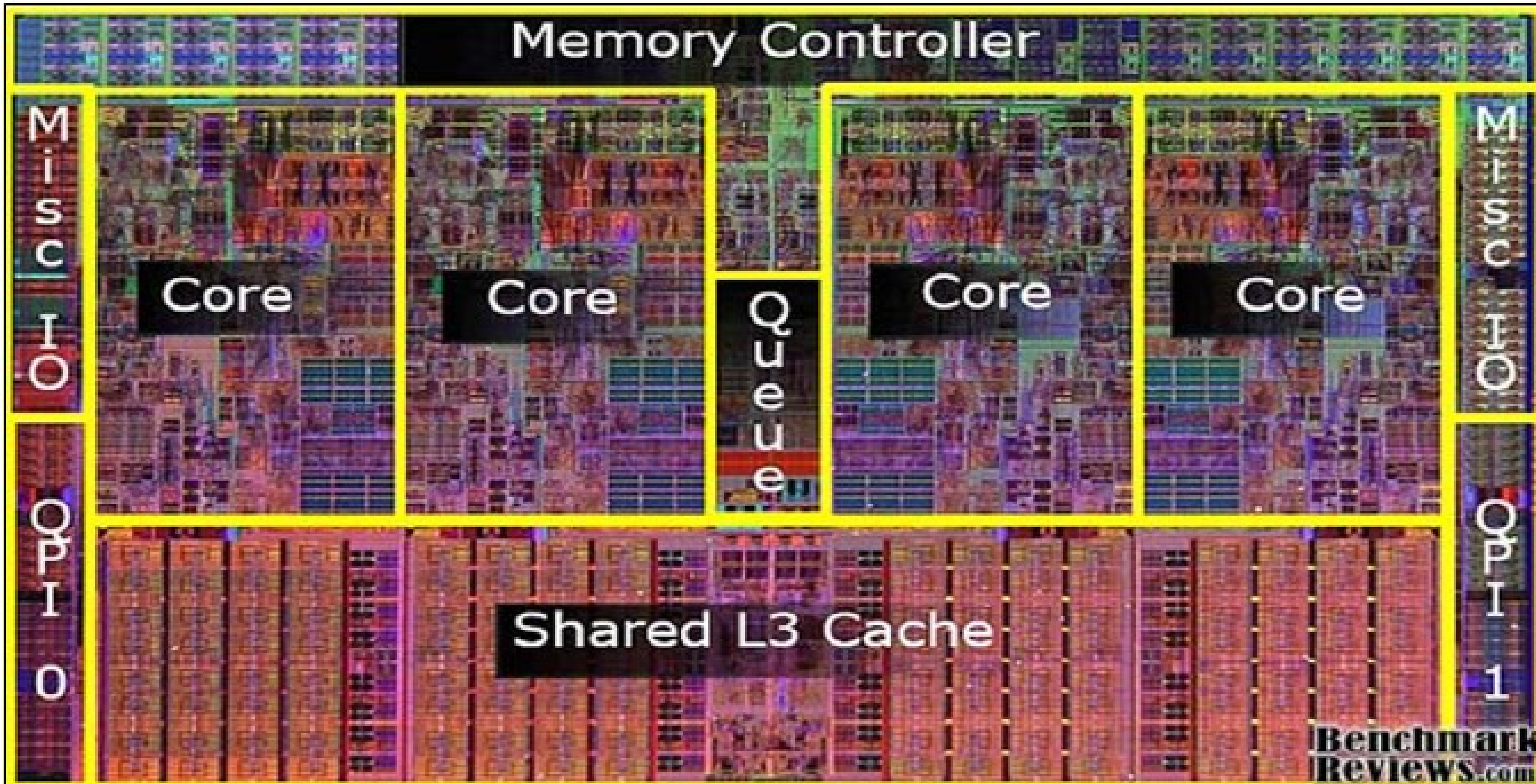
Connectivity is Critical: Bus vs Point-to-Point

- traditional design: front-side-bus (FSB)
  - each processor has to compete for access
  - multitude of processors/resources result in bottleneck

- ways forward: **localised memory** and **on-chip networks (switch)**
  - multiple simultaneous point-to-point (P2P) connections between cores & resources (much like end-to-end '**channels**')

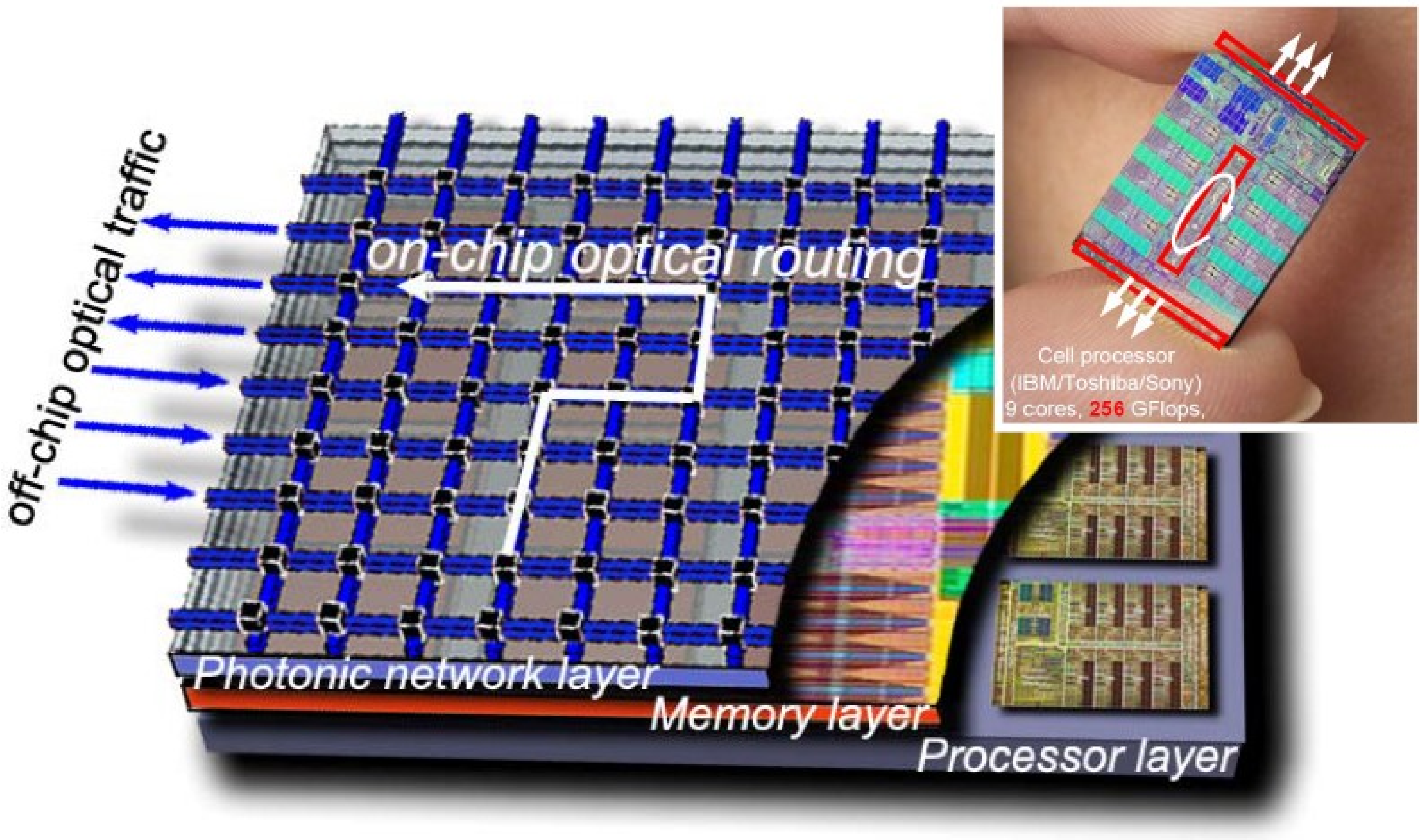


Example 1: Intel i7 Architecture

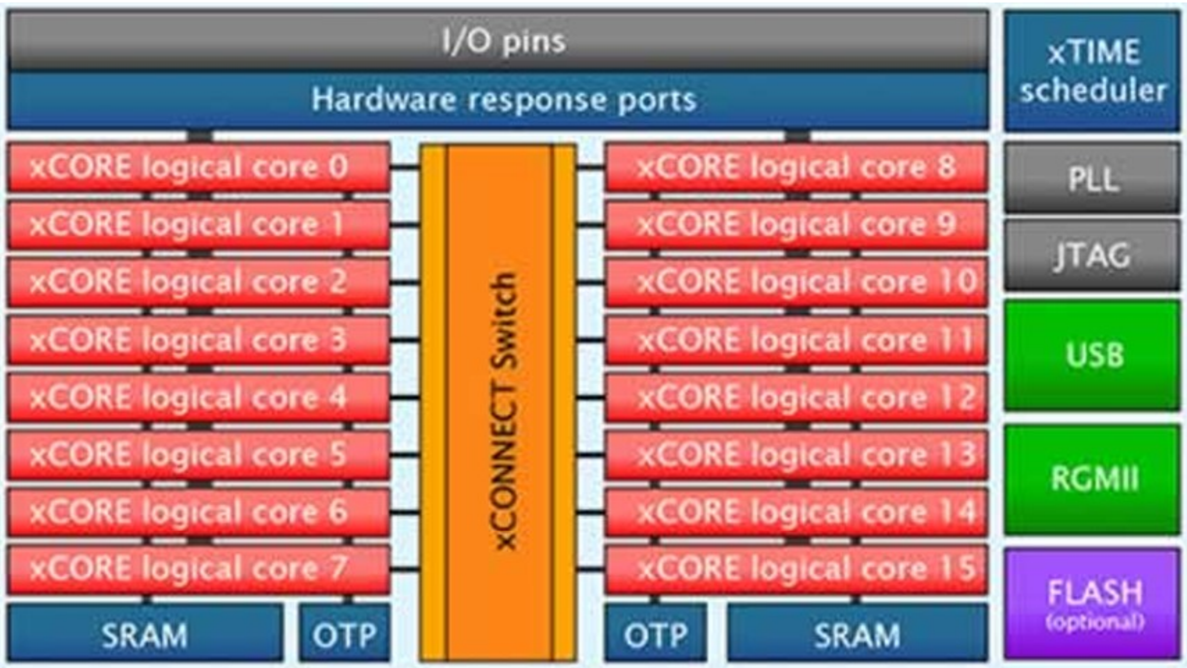


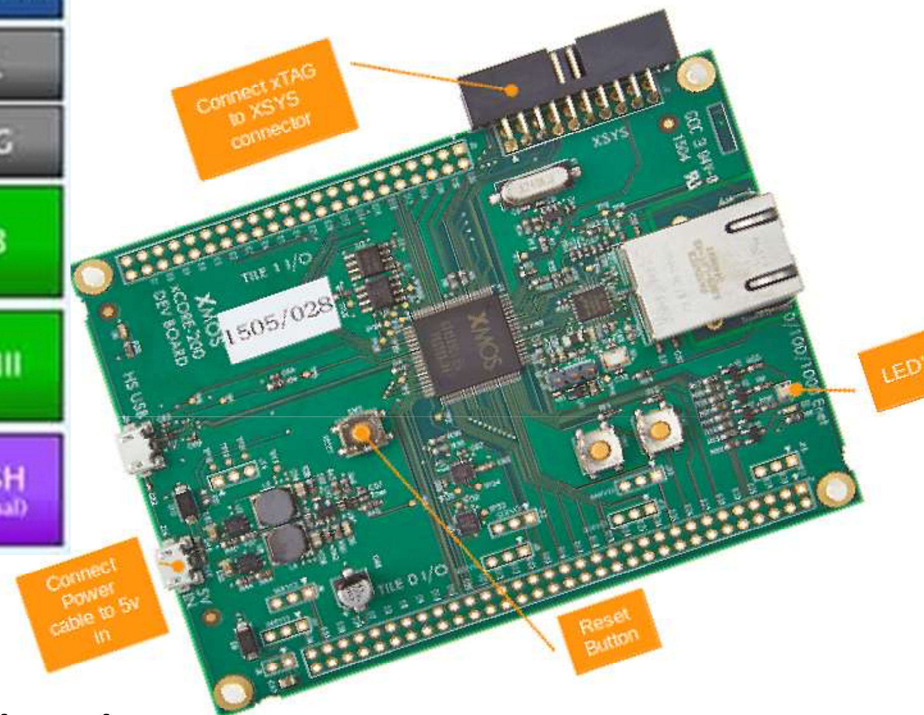
- each processor has **its own dedicated memory** using integrated memory controller (IMC)
- **P2P QuickPath onchip net** for cross-core access + snoop traffic

Example 2: Future IBM Optical On-Chip Connectivity



Example 3: XMOS xCore200 Explorer Kit

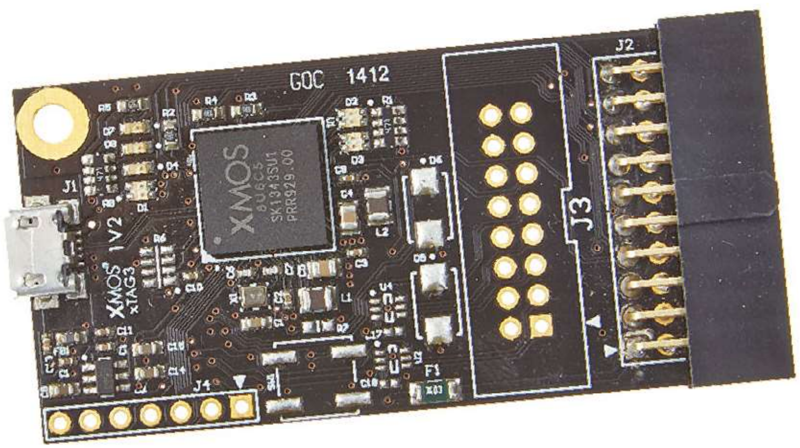




- 16 logical cores on 2 xCORE tiles
- 32 channels for cross-core communication
- 512KB internal single-cycle SRAM (max 256KB per tile)
- 6 servo interfaces, 3D accelerometer, Gigabit Ethernet interface, 3-axis gyroscope, USB interface, xTAG debug adaptor, …

Why learn xC?

- built around multi-threading and point-to-point **channel communication** model
(…thus, in line with current hardware design trends…)
- compiles directly to drive **multi-core hardware** (XMOS XS1)
- **familiar C syntax,** yet semantic similarities to classic parallel languages such as Occam
- theoretically grounded in **process algebra** CSP, which can be used to reason about (usually basic) XC programs

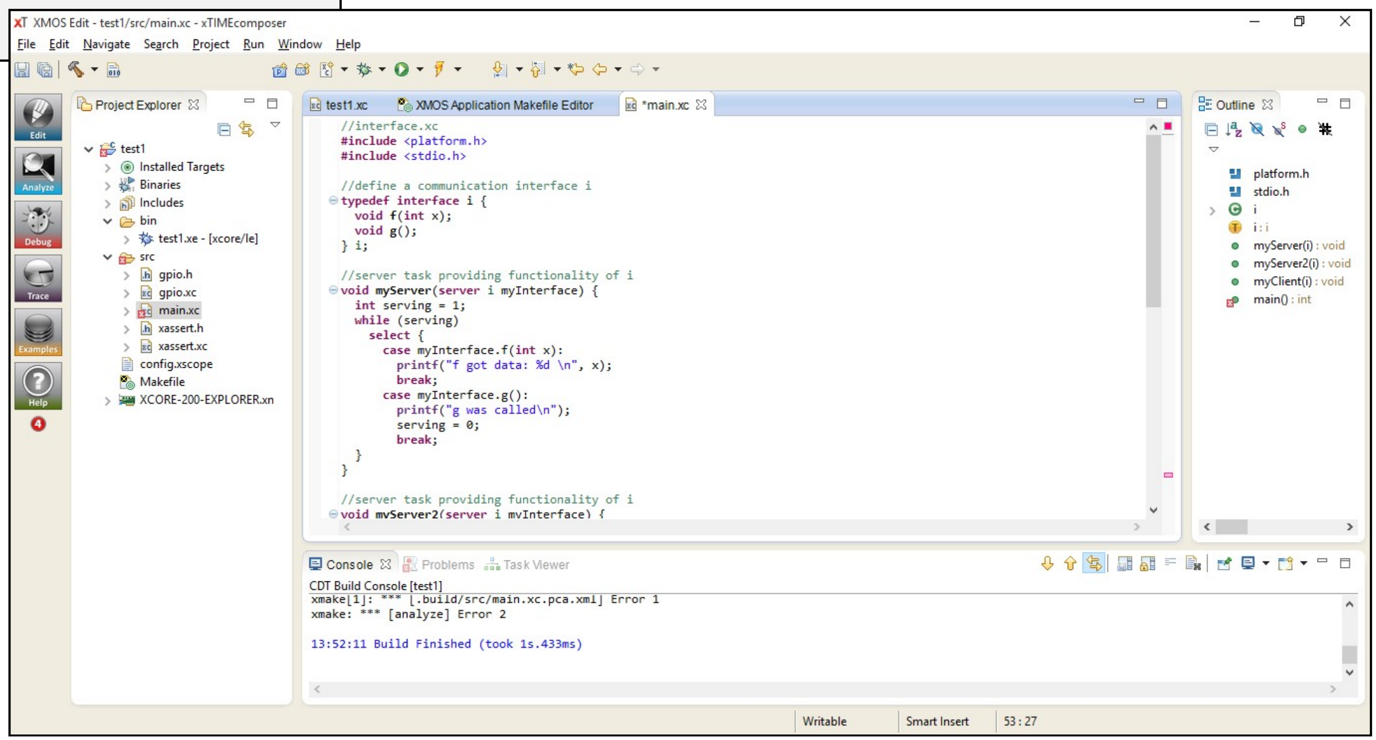


Getting Started: `hello.xc` & xTimeComposer IDE

hello.xc

```
#include <stdio.h>

int main(void) {
  printf("Hello!\n");
  return 0;
}
```

- installed in Linux lab MVB2.11
- freely available for download for Linux, Windows and MacOS (www.xmos.com/support/tools)



So far in C ... Sequential Control Flow (Deterministic)

- given an input, a single sequential process (=thread) produces a **single sequence of memory state changes** deriving the output

→ XC: every **basic block** `{ }` is treated as a sequential process with a strict order of execution (…first do this, next this …)

```
int func(int a, int b) {
  if (a > 0) {
  a -= 1;
  b *= 2;
  }
  a *= b;
  return a;
}
```

XC example: sequential process

```
func(2,4)>>

1:   a = 2; b = 4;
2:   a = 1; b = 4;
3:   a = 1; b = 8;
4:   a = 8; b = 8;

>> 8
```

its memory trace for input (2,4)

Where C alone cannot go – a wish list...

- EXPLICIT PARALLELISM

  we want to execute several statements in parallel on different cores to gain a speed advantage over sequential execution
  (…trading temporal spread for spatial spread of computation)

- EXPLICIT COMMUNICATION

  we want to channel messages between cores/threads to synchronise several concurrent computations

- EXPLICIT CONTROL

  we want to control the physical location of execution and storage to minimise data transfers and effectively use local resources (…compactness under programmer control)

xC Concurrent Execution: **PAR** statement

```
int func(int a, int b) {
   if (a > 0)
     par {
        a -= 1;
        b *= 2;
     }
   a *= b;
   return a;
 }
```

par (parallel statement) …execute each block within body concurrently…

…each statement of **par**-body is (potentially) executed in parallel by starting a separate thread on a free core…

…wait here until all statements in body have returned…

Non-deterministic Control Flow

- given an input, a set of concurrent processes (=threads) produces **one out of many possible sequences** of memory state changes deriving an output ('implicit choice' during runtime)

→ XC: every statement/sub-block in a `par{}` block is treated as an **independent** process

```
int func(int a, int b) {
  if (a > 0)
    par {
      a -= 1;
      b *= 2;
    }
  a *= b;
  return a;
}
```

```
func(2,4)>>

1: a = 2; b = 4;
2: a = 1; b = 4;
3: a = 1; b = 8;
4: a = 8; b = 8;

>> 8
```

trace sequence 1

```
func(2,4)>>

1: a = 2; b = 4;
2: a = 2; b = 8;
3: a = 1; b = 8;
4: a = 8; b = 8;

>> 8
```

trace sequence 2

Program Example: Execution in Parallel

par.xc

```
// par.xc
#include <platform.h>
#include <stdio.h>

void hello(int threadNo);

// main starting two tasks in parallel
int main(void) {
  par {
    hello(0); //start first thread in parallel
    hello(1); //start second thread in parallel
  } // wait until both threads have terminated
  return 0;
}

// function to print message
void hello(int threadNo){
  printf("Hello from thread #%d.\n", threadNo);
}
```

Key Concept: Process Construction

Two ways of combining processes into a compound process:

- **SEQUENTIAL concatenation** in a block {}
  - returns when its last component process finishes
  - executed one after the other (…implicit in XC…)

- **PARALLEL composition** in a block par {}
  - written order of components is irrelevant
  - returns when all its component processes have returned

Any process, compound or just a single statement, …

- **starts** (i.e. thread is instantiated),
- **performs** a number of actions (i.e. thread runs)
- and then **may finish/terminate** (i.e. thread returns to caller)

Revisited: Concurrency vs Parallelism

CONCURRENCY…
…is concerned with **non-deterministic composition** of processes (i.e. program components)

PARALLELISM…
…is concerned with **exploiting independencies** among the sub-computations of a deterministic computation

NO COMPILER… is available today that automatically turns a sequential process into a set of communicating, concurrent processes that optimally exploit independencies among the sub-computations.

→ programmers need to understand concurrent programming paradigm to exploit & support emerging physical parallelism

Example: Execution on Different Physical Tiles

hellotile.xc

```
// hellotile.xc
#include <platform.h>
#include <stdio.h>

void hello(int tileNo);

// main starting two tasks in parallel on different tiles
int main(void) {
  par {
    on tile[0] : hello(0); //start on tile 0
    on tile[1] : hello(1); //start on tile 1
  }
  return 0;
}

// function to print message
void hello(int tileNo){
  printf("Hello from tile #%d.\n", tileNo);
}
```

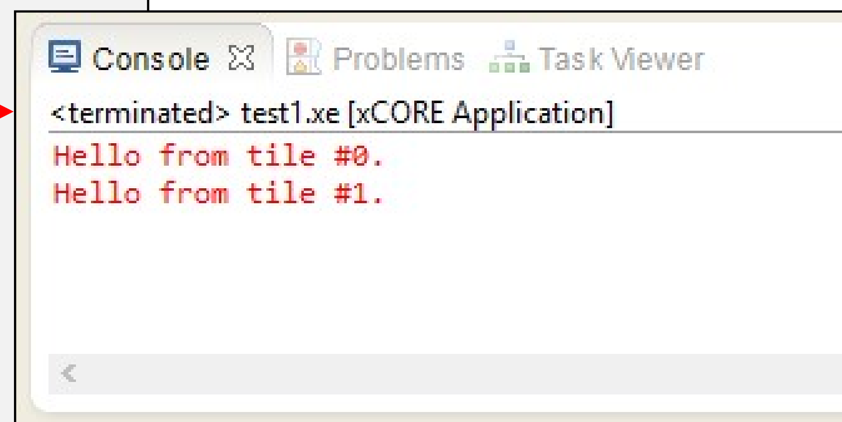
Example: Combining XC with C Sources & Timers

partxc.xc

```
//partxc.xc
#include <platform.h>
#include <stdio.h>

extern void hello(int tileNo);

int main(void) {
  par {
    on tile[1] : hello(1);
    on tile[0] : hello(0);
  }
  return 0;
}

//delays execution
void delay(uint delay)
{
    uint time, tmp;
    //define a timer
    timer t;
    //read current state of timer
    t :> time;
    //trigger when timer has moved on the delay no of ticks
    t when timerafter ( time + delay ) :> tmp;
}
```

partc.c

```
//partc.c
#include <stdio.h>
#include <platform.h>

extern void delay(uint delay);

void hello(int tileNo){
  delay((3-tileNo)*1000);
  printf("Hello from tile #%d.\n",tileNo);
}
```

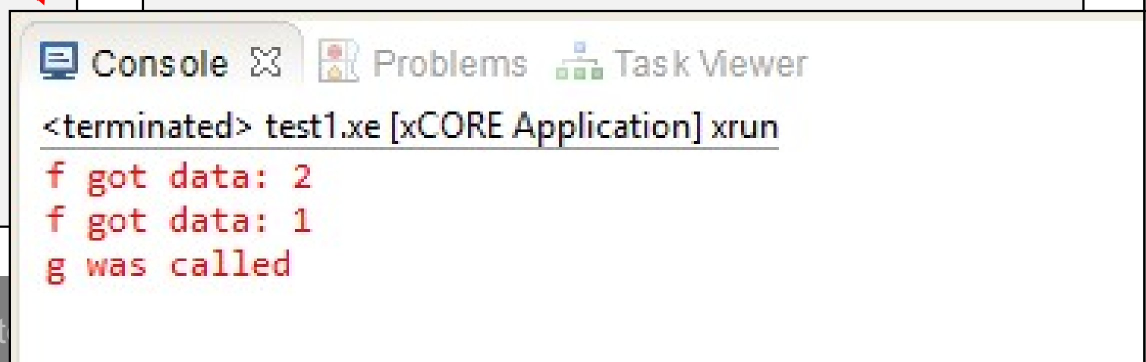


Example: Interfaces for Single Client-Server Setups

interface.xc

```
//interface.xc
#include <platform.h>
#include <stdio.h>

//define a communication interface i
typedef interface i {
  void f(int x);
  void g();
} i;

//server task providing functionality of i
void myServer(server i myInterface) {
  int serving = 1;
  while (serving)
   select {
    case myInterface.f(int x):
      printf("f got data: %d \n", x);
      break;
    case myInterface.g():
      printf("g was called\n");
      serving = 0;
      break;
} } ...
```

```
...

//client task calling function
//of task 2
void myClient(client i myInterface) {
  myInterface.f(2);
  myInterface.f(1);
  myInterface.g();
}

//main starting two threads
//calling over an interface
int main() {
  interface i myInterface;
  par {
    myServer(myInterface);//only1server
    myClient(myInterface);//only1client
  }
  return 0;
}
```



Lecture 3 Outlook

Channel Communication