High Performance Computing

3. Hardware Platforms for High-Performance Embedded Computing

Andrea Marongiu
(andrea.marongiu@unimore.it)
AA 2018-2019

Recall from previous class

The Dim Horseman (#2)

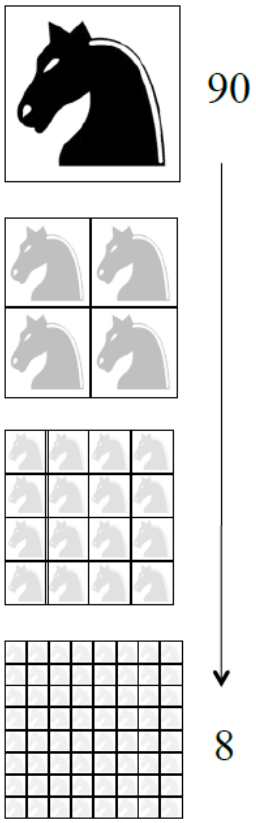
Dark silicon

↓

Multicores

"We will fill the chip with homogeneous cores that would exceed the power budget but we will underclock them (spatial dimming), or use them all only in bursts (temporal dimming)

… "dim silicon".



Spatial Dimming

- Gen1&2 multicores (higher core counts → lower freqs)
- Near-threshold voltage operation

Temporal Dimming

Thermally limited systems

- **ARM Big-little** (A15 power usage way above sustainable for phone → 10sec burst at most)
- Battery-limited systems
- Quad-core mobile application processor

Recall from previous class

The Specialized Horseman (#3)

Dark silicon

Heterogeneous Systems-on-Chip (SoC)

"We will use all of that dark silicon area to build specialized cores, each of them tuned for the task at hand (10-100x more energy efficient), and only turn on the ones we need…"

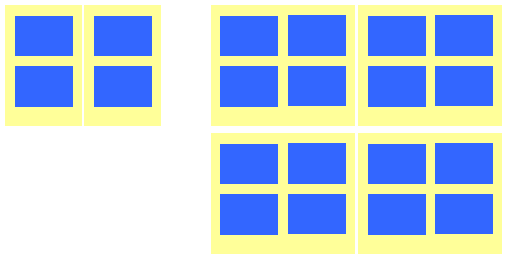
[e.g., Venkatesh et al., ASPLOS 2010,
Lyons et al., CAL 2010,
Goulding et al., Hotchips 2010,
Hardavellas et al. IEEE Micro 2011]

Specialization is the goal behind architectural heterogeneity

Evolution of multicores

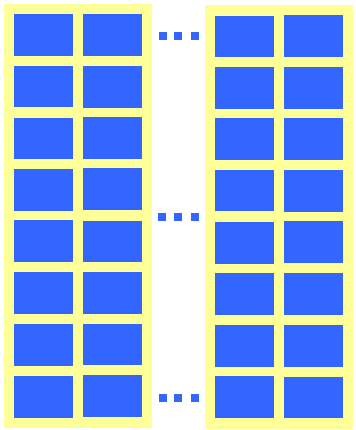
Classification of multicore processors according to the layout of their CPU cores

- **Multicores with homogeneous CPU cores**
  - **Traditional MC processors** — 2 ≤ n ≈≤ 32 cores
    - **Mobiles/ desktops**
    - **Servers**



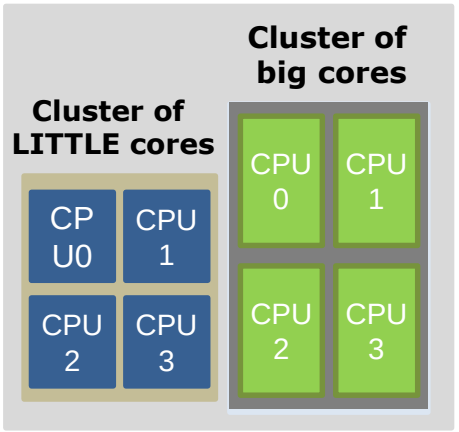
Mainstream computing (since 2001-2006)
Mobiles (2006)

  - **Manycore processors** — with n ≈> 32 cores



Experimental (2007-2010) production systems, Intel's Xeon Phi (2012)

- **Multicores with heterogeneous CPU cores**
  - **big.LITTLE processors**



Mobiles (since 2011)

Evolution of multicores

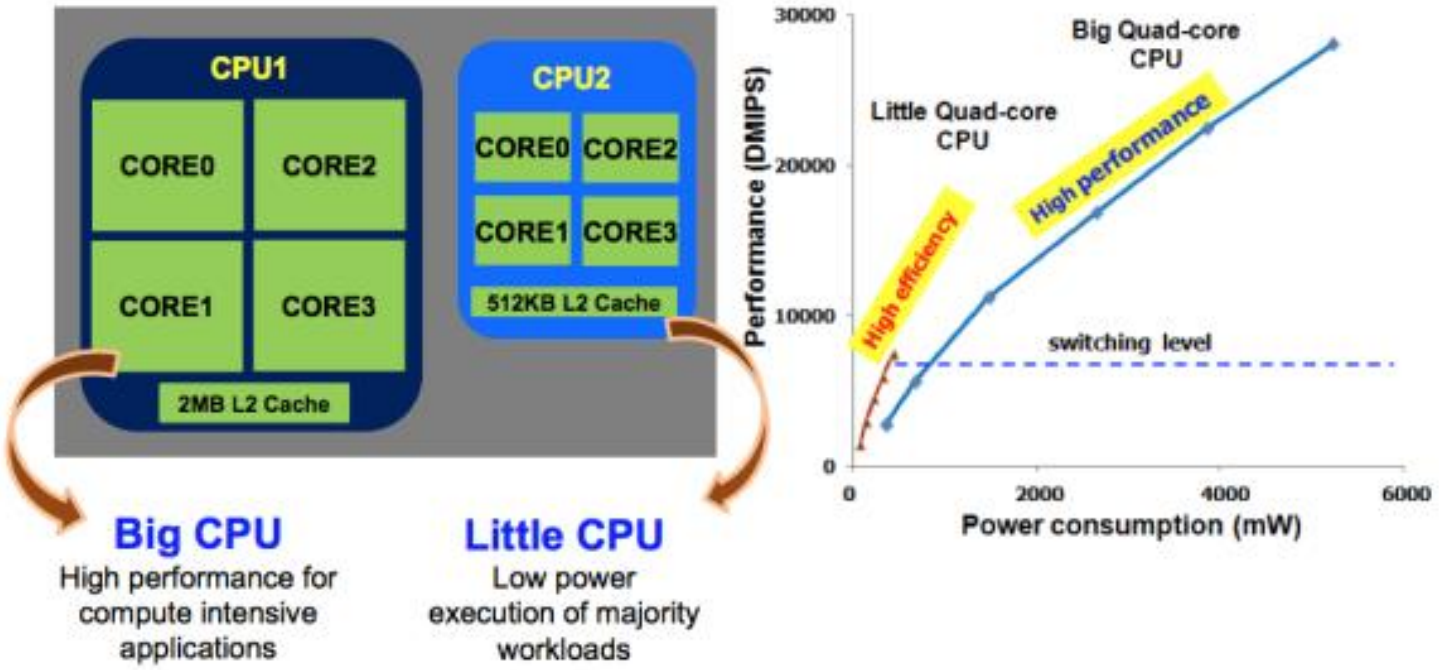
Multicore and manycore processors

With core counts exceeding certain limits, e.g. recently 16 or 32 cores, some architectural subsystems become incapable to suitable support the increased number of cores, e.g.

- to provide high enough memory bandwidth or
- to provide a fast enough core to core communication.

Therefore, such processors need a novel microarchitectures and will be typically called **manycore processors** to distinguish them from traditional built multicore processors.

Multicores for mobile: Big-Little

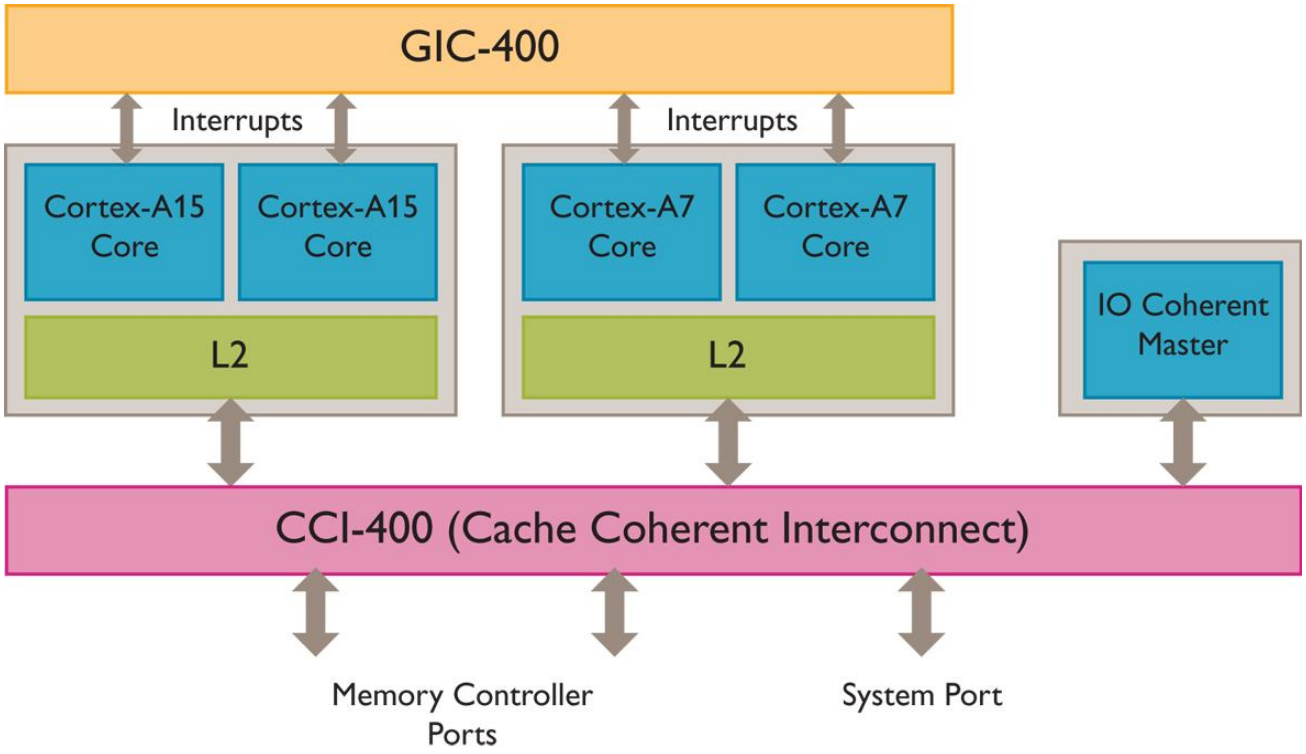


A mobile with heterogeneous CPU cores: Samsung Exynos 5 Octa 5410 in big.LITTLE configuration (2013 revealed)

Principle of operation:

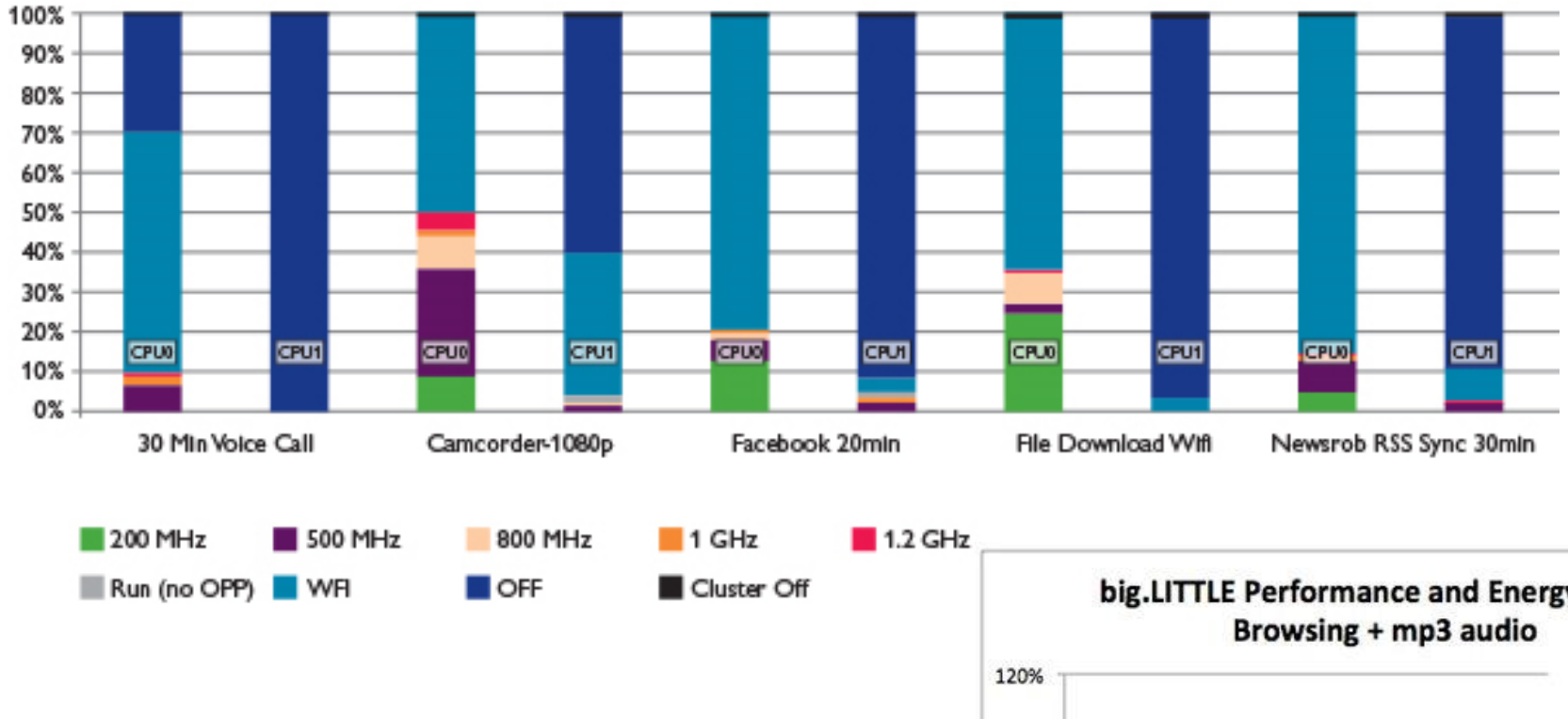
The big or the LITTLE core cluster is allocated for a task according to its performance demand, the cluster of big cores is allocated to compute intensive tasks whereas the cluster of LITTLE cores to less demanding tasks.

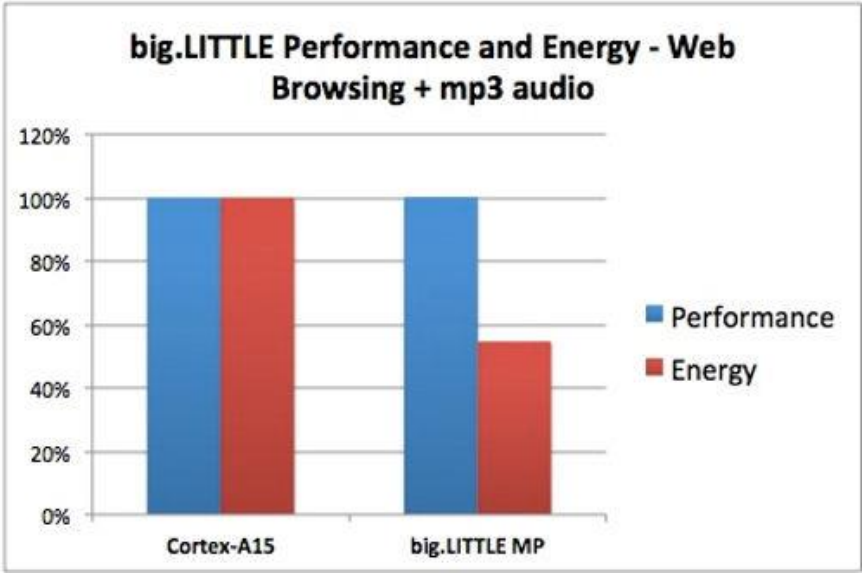
ARM big.LITTLE



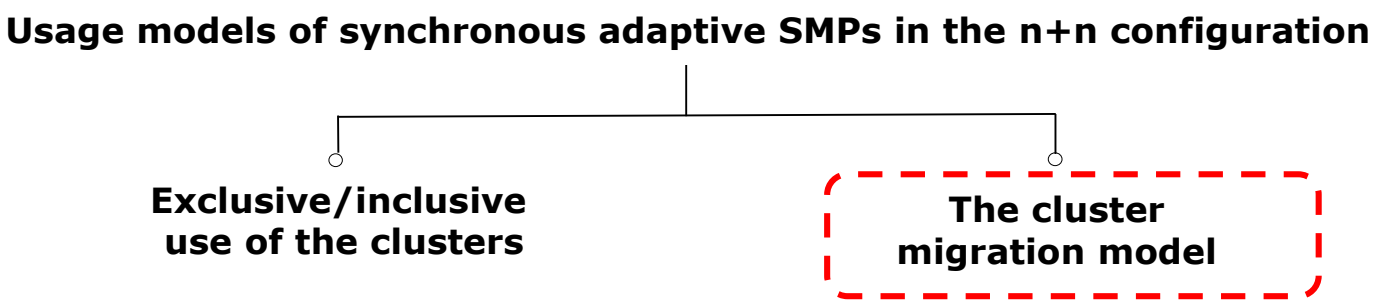
- DVFS (Dynamic Voltage and Frequency Scaling)
- Task Migration
- SW Overhead
- Memory coherency

ARM big.LITTLE Energy Saving





Principle of ARM big.LITTLE technology



Principle of ARM big.LITTLE technology

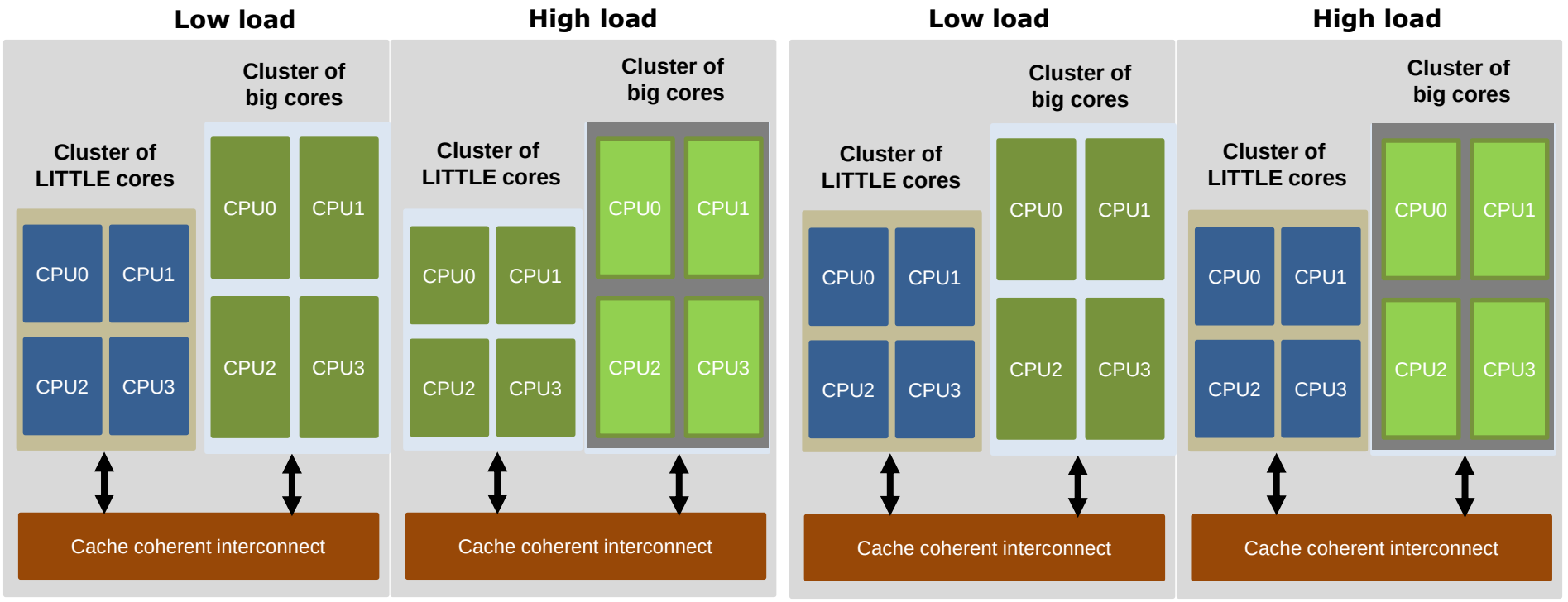
Exclusive/inclusive use of the clusters

Exclusive use of the clusters

Clusters are used exclusively,
i.e. at a time one of the clusters is in use
as shown below for the cluster migration model
(to be discussed later)

Inclusive use of the clusters

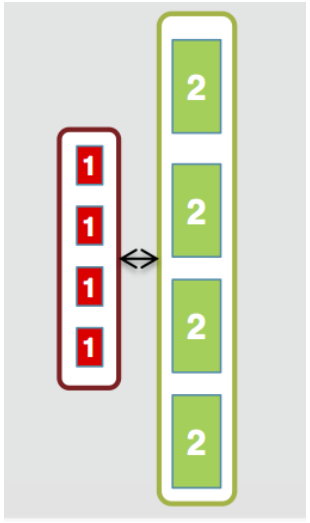
Clusters are used inclusively,
i.e. at a time both clusters can be used
partly or entirely



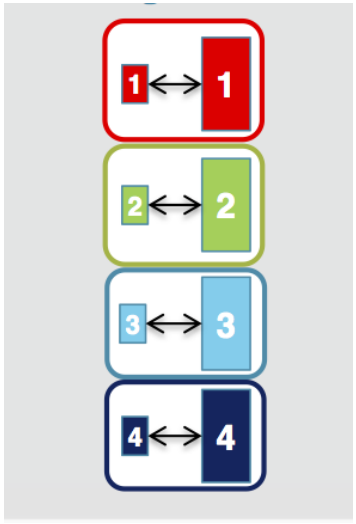
Principle of ARM big.LITTLE technology

The cluster migration model

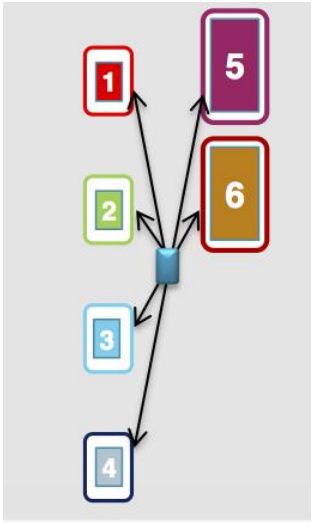
- **Exclusive use of the clusters**
  - **Cluster migration**
  - **Core migration**
- **Inclusive use of the clusters**
  - **Core migration**



big.LITTLE processing with cluster migration



big.LITTLE processing with core migration

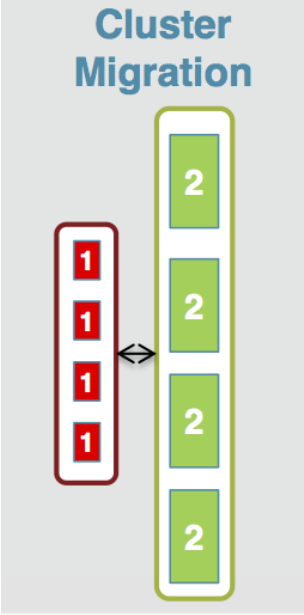


big.LITTLE MP

Principle of ARM big.LITTLE technology

Big.LITTLE processing with cluster migration [5]

- There are two core clusters, the LITTLE core cluster and the big core cluster.
- Tasks run on either the LITTLE or the big core cluster, so only one core cluster is active at any time (except a short interval during a cluster switch).
- Low workloads, such as background synch tasks, audio or video playback run typically on the LITTLE core cluster.
- If the workload becomes higher than the max performance of the LITTLE core cluster the workload will be migrated to the big core cluster and vice versa.

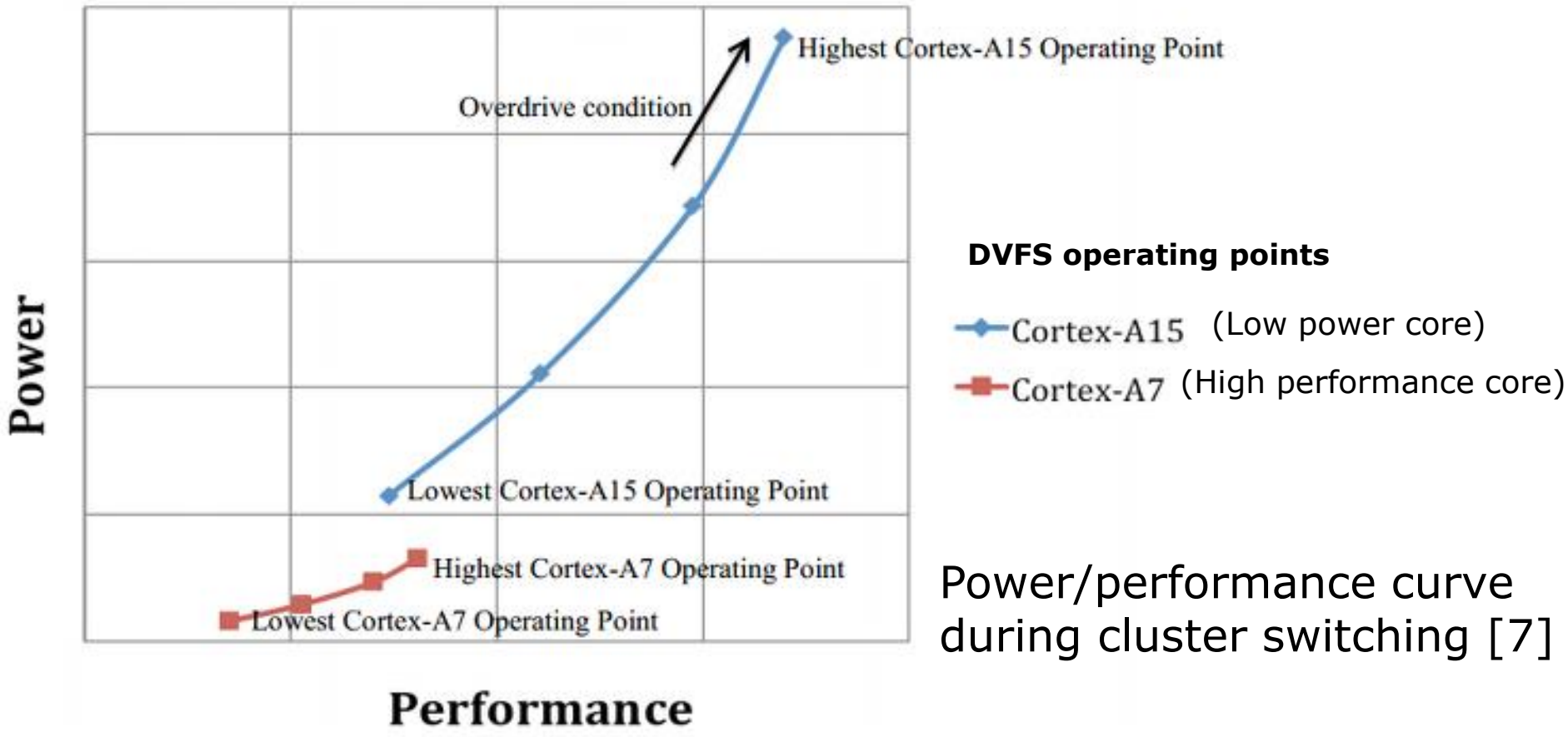


Principle of ARM big.LITTLE technology

Cluster switches [6]

- Cluster selection is driven by OS power management.
- OS (e.g. the Linux cpufreq routine) samples the load for all cores in the cluster and selects an operating point for the cluster.
- It switches clusters at terminal points of the current clusters DVFS curve, as illustrated in the next Figure.

Principle of ARM big.LITTLE technology



Power/performance curve during cluster switching [7]

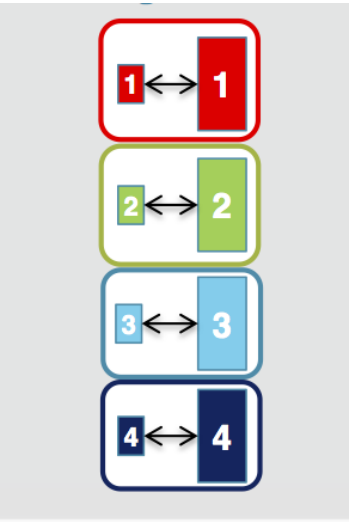
- A switch from the low power cluster to the high performance cluster is an extension of the DVFS strategy.
- A cluster switch lasts about 30 kcycles.

Principle of ARM big.LITTLE technology

Big.LITTLE processing with core migration [5], [8]

- There are two core clusters, the LITTLE core cluster and the big core cluster.
- Cores are grouped into pairs of one big core and one LITTLE core.

  The LITTLE and the big core of a group are used exclusively.
- Each LITTLE core can switch to its big counterpart if it meets a higher load than its max. performance and vice versa.
- Each core switch is independent from the others.



Principle of ARM big.LITTLE technology

Core switches [6]

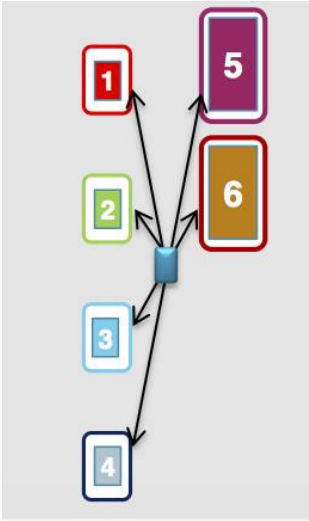
- Core selection in any core pair is performed by OS power management.
- The DVFS algorithm monitors the core load.

  When a LITTLE core cannot service the actual load, a switch to its big counterpart is initiated and the LITTLE core is turned off and vice versa.

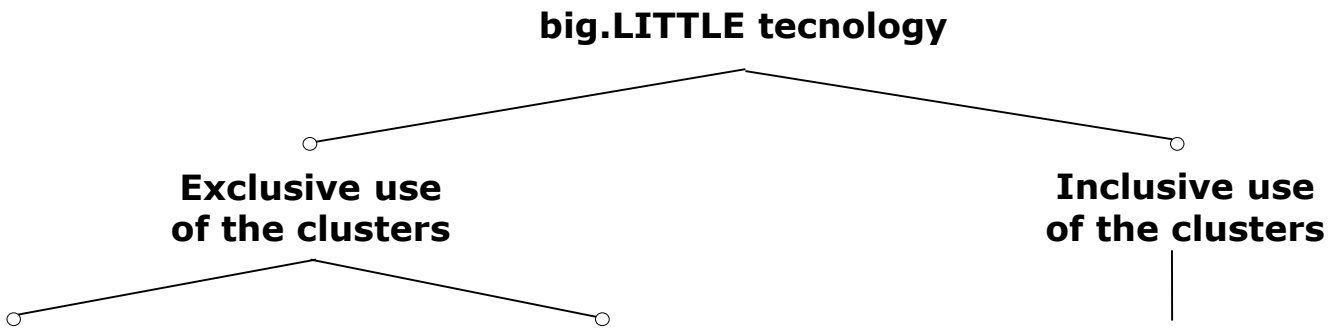
Principle of ARM big.LITTLE technology

big.LITTLE MP processing with core migration [8],[5]

- The OS scheduler has all cores of both clusters at its disposal and can activate all cores at any time.
- Tasks can run or be moved between the LITTLE cores and the big cores as decided by the scheduler.
- big.LITTLE MP termed also as Heterogeneous Multiprocessing (HMP).



big.LITTLE in recent mobile processors



big.LITTLE processing with cluster migration

Described first in ARM's White Paper (2011) [3]

Used in
Samsung Exynos 5 Octa 5410 (2013) (4 + 4 cores)

big.LITTLE processing with core migration

Described first in ARM's White Paper (2012) [9]

big.LITTLE MP (Heterogeneous Multiprocessing)

Described first in ARM's White Paper (2011) [3]

Renesas MP 6530 (2013) (2 + 2 cores)

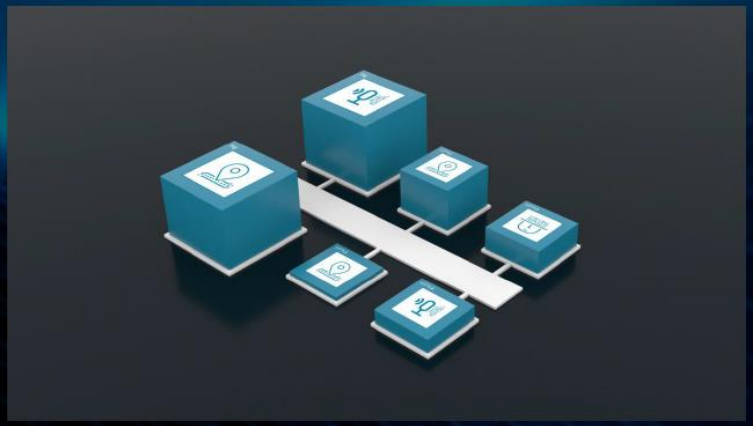
Samsung HMP on Exynos 5 Octa 5420 (2013) (4 + 4 cores)

Qualcomm Snapdragon 835 (MSM8998) (2016) (4 + 4 cores)

big.LITTLE reloaded: DynamIQ

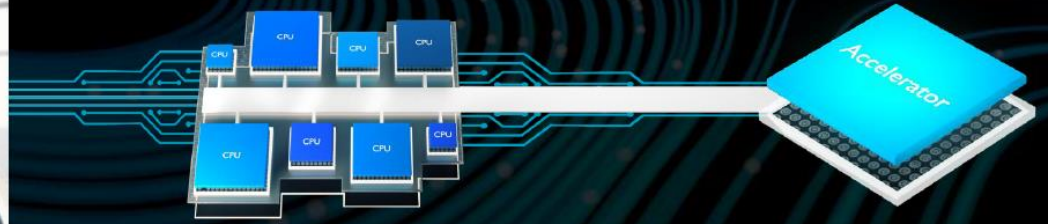
- A new single cluster design for big.LITTLE
- Increased efficiency from shared memory between CPUs
- Higher performance through faster task migration

DynamIQ big.LITTLE



DynamIQ boosting AI/ML performance both on CPU and in system

Dedicated processor instructions for AI



Improved access to acceleration

More than **50x AI performance** boost on the CPU in the next 3-5 years

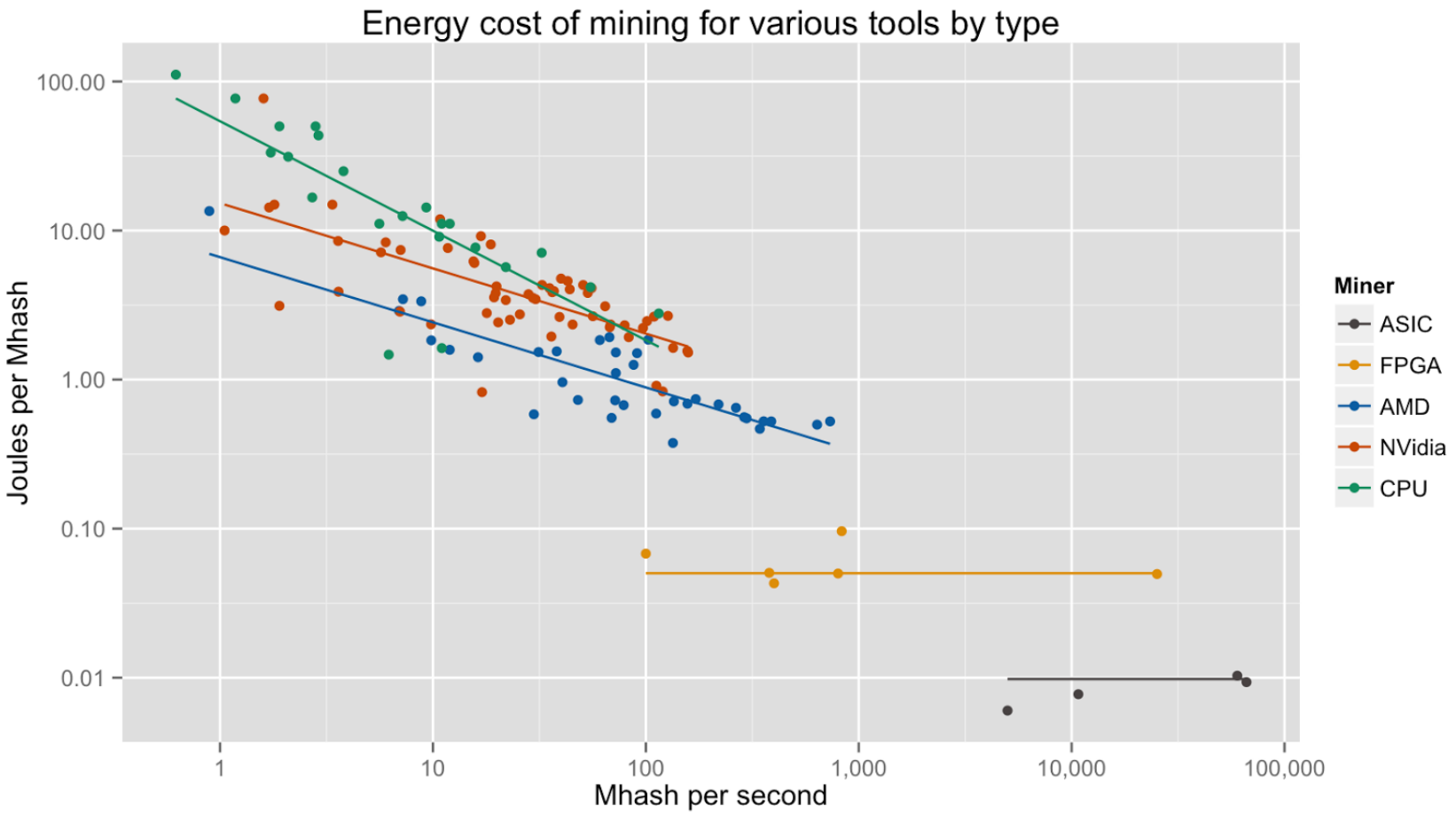
Up to **10x quicker** response to accelerators

Heterogeneous SoCs

Heterogeneous SoCs

- DynamIQ is not only about a multicore CPU
- Specialization is key
- **Accelerators** everywhere
  - Many ways to build accelerators…

Energy Efficiency vs. Flexibility



The more specialized the more efficient…

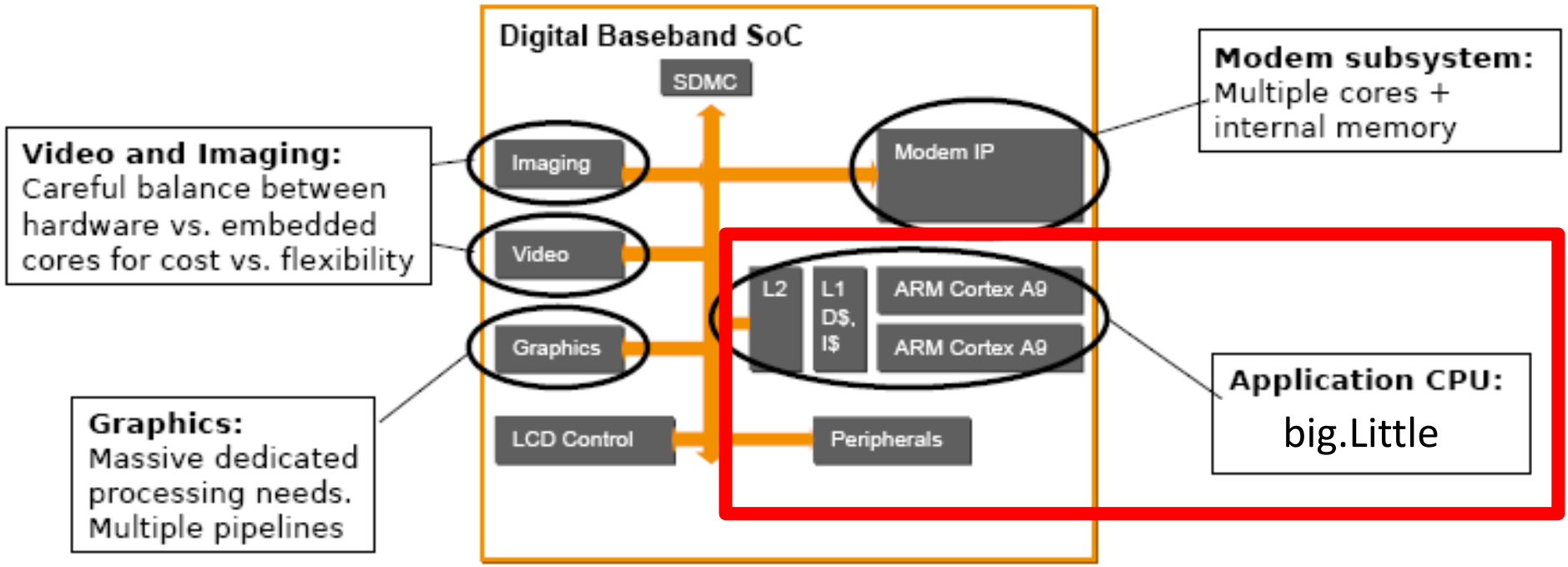
But design has a (large) cost

Source: Dr. Handel Jones, IBS "Factors for Success in System IC Business and Impact on Business Model", Q4 2012 Report.
Design costs shown based on complex, primary designs. Less complex derivative designs have lower cost (nominally ~ 40% of the cost shown for complex primary designs).
Socket volume assumptions: 20% of Revenue on R & D and $20 average selling price per unit

Programmable solutions dominate…

Integrated SoC Mobile

- High-speed SMP for “almost sequential” GP
- “Processor arrays” for domain-specific throughput computing (100x GOPS/W) ultra parallel…



H-SOC in 2016/17 Tegra X1

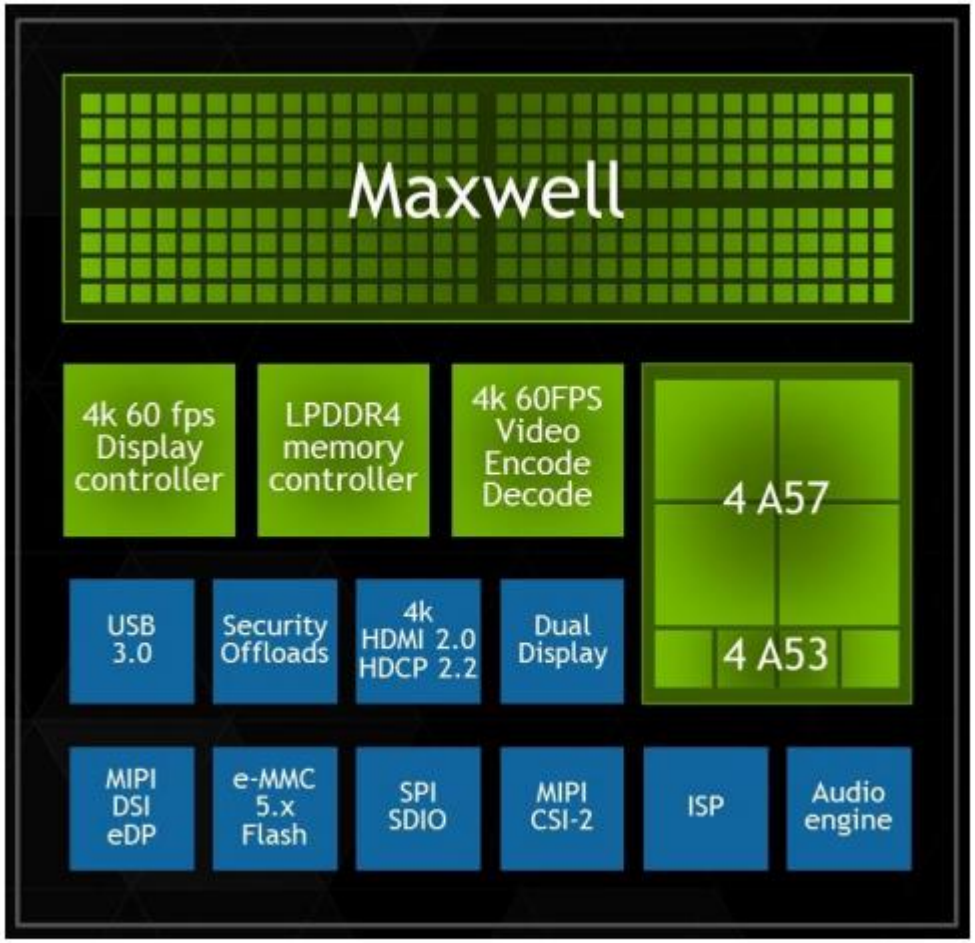
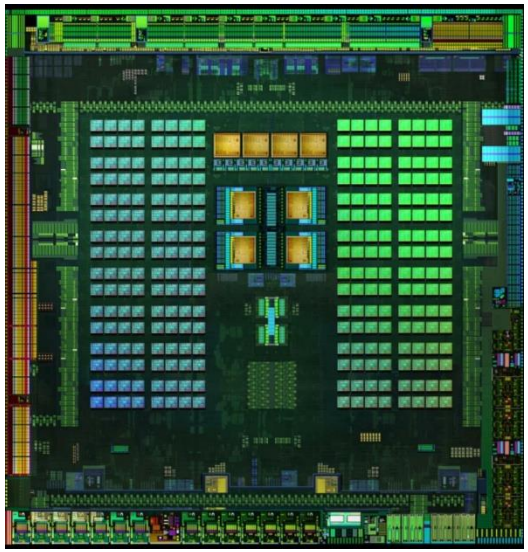


Figure 1 NVIDIA Tegra X1 Mobile Processor



TEGRA X1 CPU CONFIGURATION

4 HIGH PERFORMANCE A57 BIG CORES

- 2MB L2 cache
- 48KB L1 instruction cache
- 32KB L1 data cache

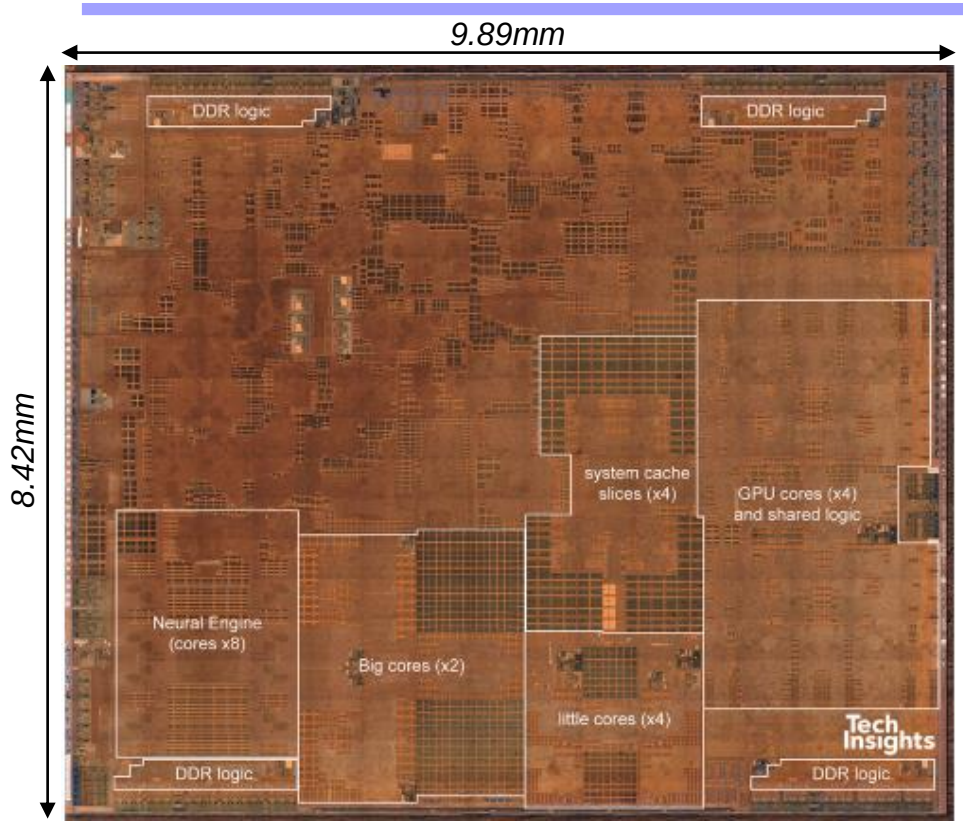
4 HIGH EFFICIENCY A53 LITTLE CORES

- 512KB L2 cache
- 32KB L1 instruction cache
- 32KB L1 data cache

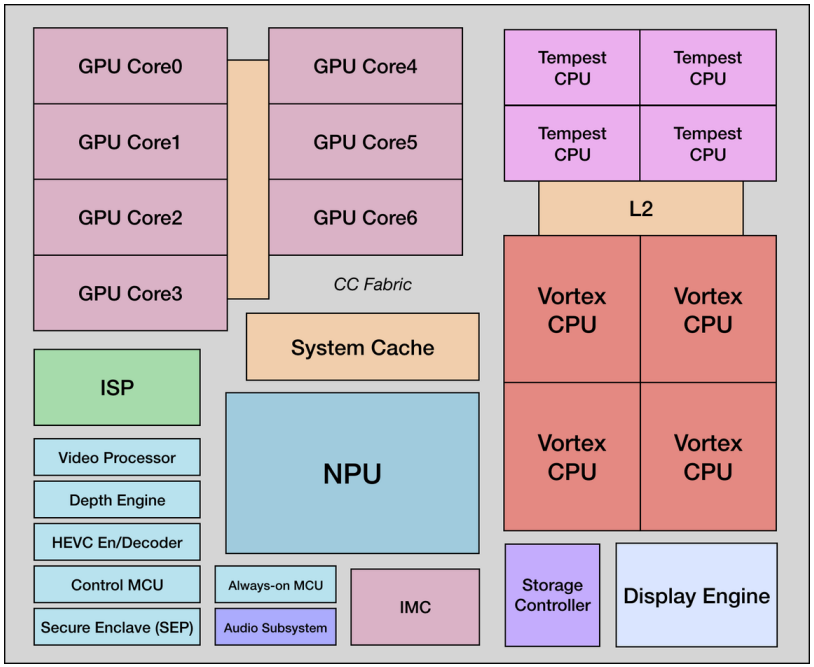
Tegra X1 GPU

GPU	Tegra K1 (Kepler GPU)	Tegra X1 (Maxwell GPU)
SMs	1	2
CUDA Cores	192	256
GFLOPs (FP32) Peak	365	512
GFLOPs (FP16) Peak	365	1024
Texture Units	8	16
Texel fill-rate	7.6 Gigatexels/sec	16 Gigatexels/sec
Memory Clock	930 MHz	1.6GHz MHz
Memory Bandwidth	14.9 GB/s	25.6 GB/s
ROPs	4	16
L2 Cache Size	128KB	256KB
Manufacturing Process	28-nm	20-nm
Z-cull	256 pixels/clock	256 pixels/clock
Raster	4 pixels/clock	16 pixels/clock
Texture	8 bilinear filters/clock	16 bilinear filters/clock
ZROP	64 samples/clock	128 samples/clock

H-SOC in 2018 – Apple A12/A12X



A12 (iPhone XS) – 7nm



A12X (iPad Pro 2018)

<u>How many processors? A lot!</u>

CPU: 2/4 “big” cores + 4 “small” cores

GPU: 4/6 cores

Accelerators: Neural Processing Unit (NPU) + Image Signal Processor (ISP)

MCUs: Control + Always-On

GPU acceleration

Why GPUs?

- Graphics workloads are embarrassingly parallel
  - Data-parallel
  - Pipeline-parallel
- CPU and GPU execute in parallel
- Hardware: texture filtering, rasterization, etc.

Data Parallel

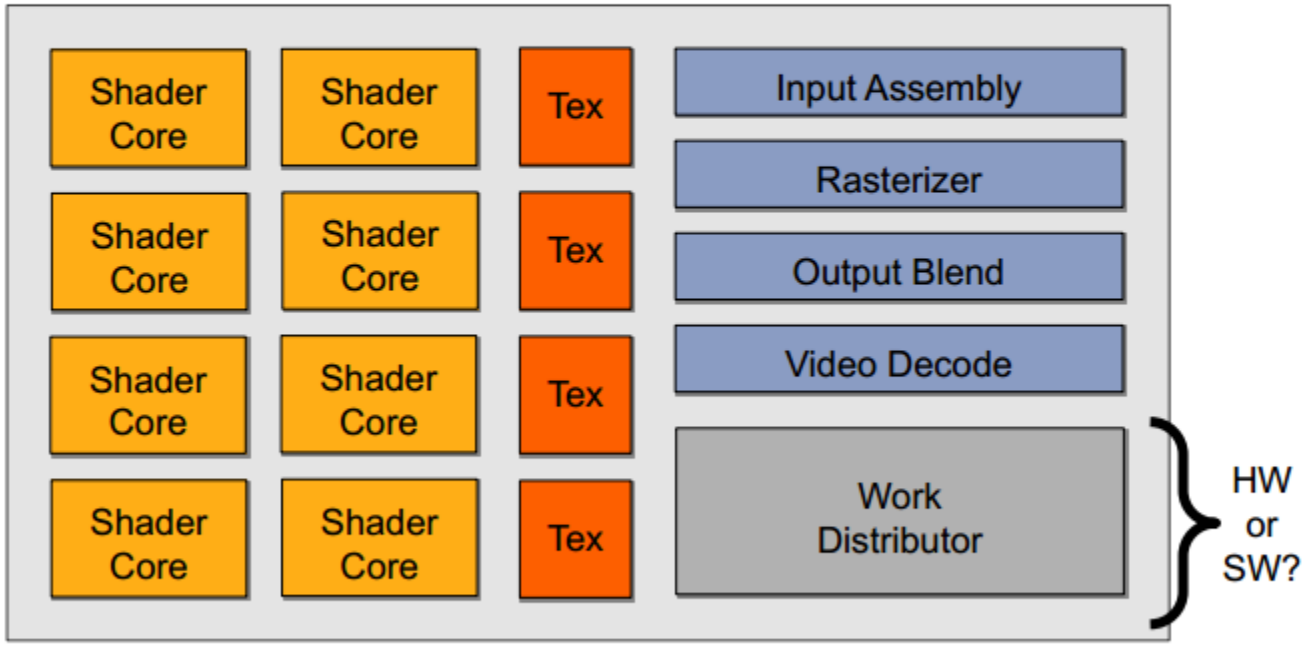
- *But remember:*
  - Many workloads from real-life applications have abundant parallelism
- *Beyond Graphics*
  - Machine vision
  - (Matrix multiply)

General-Purpose GPU (GPGPU)

GPU

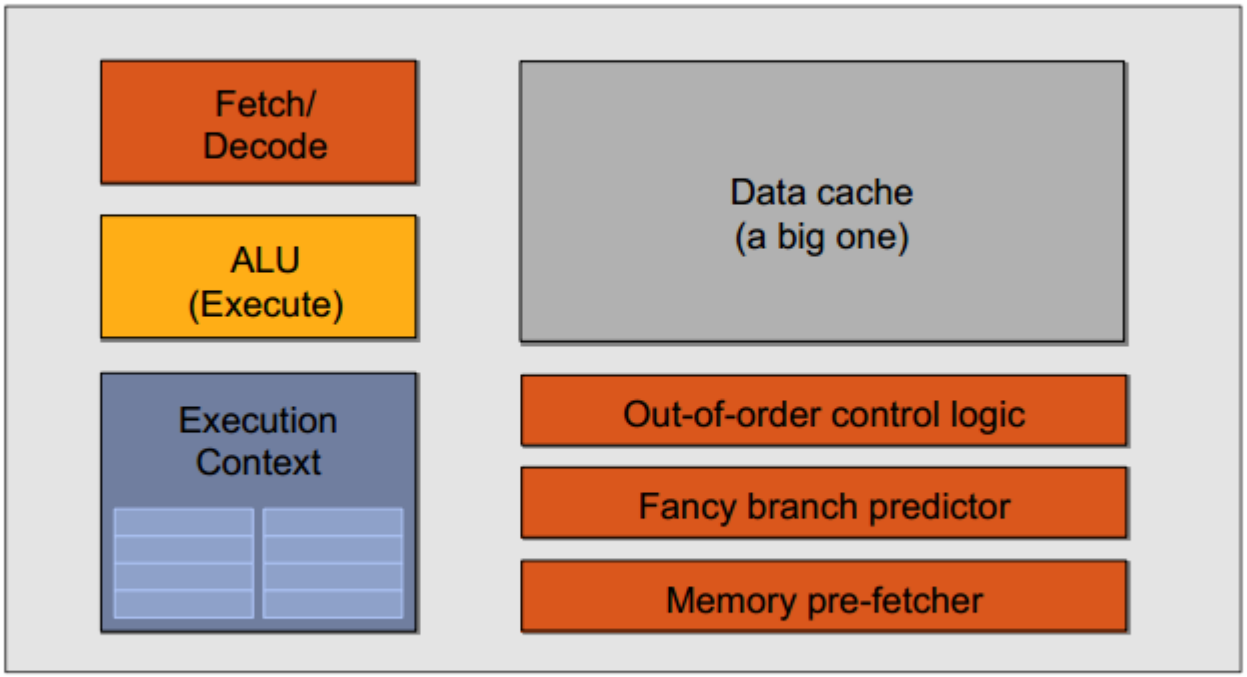
What's in a GPU?

A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)



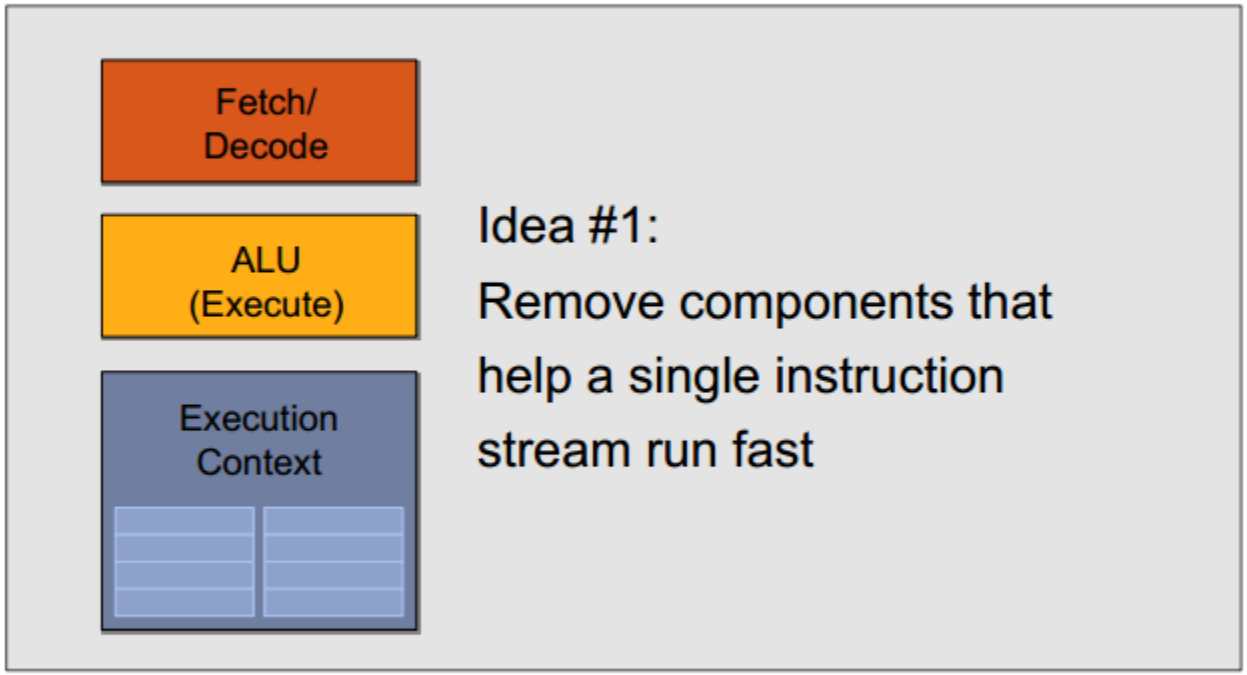
GPU

"CPU-style" cores



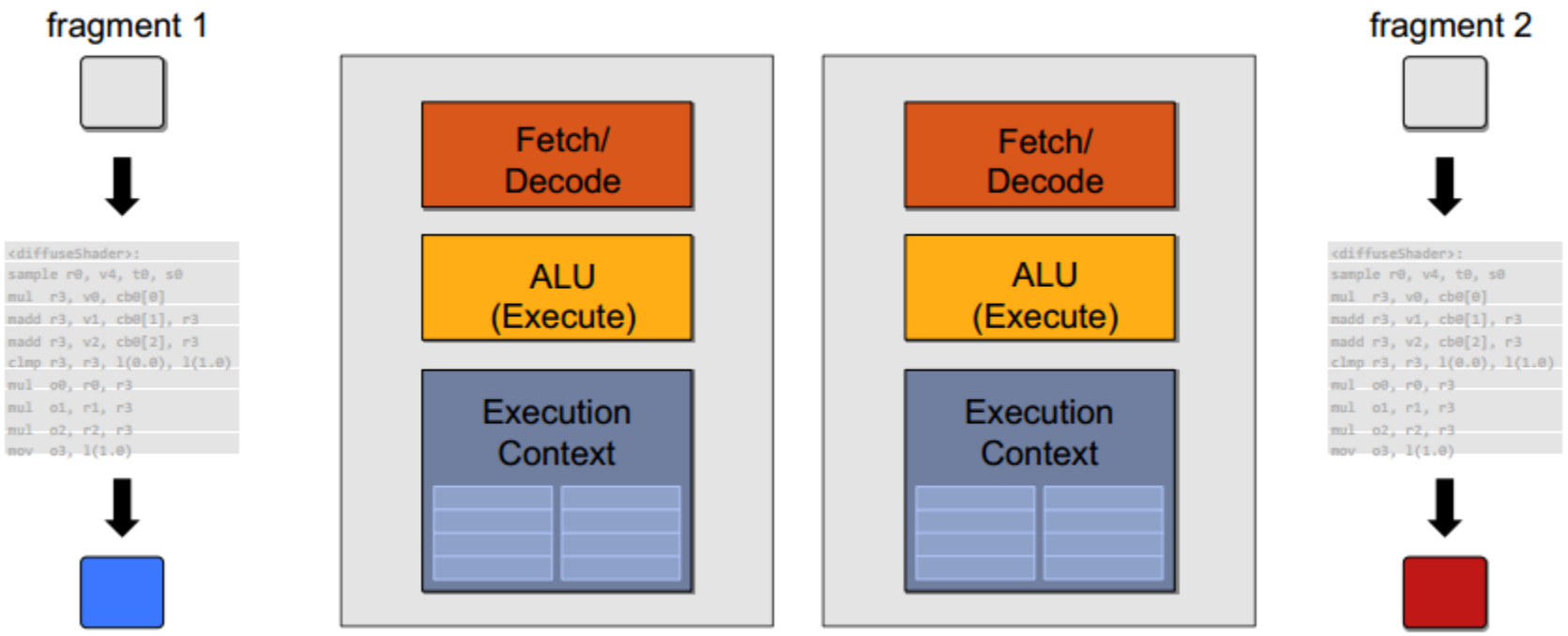
GPU

Slimming down



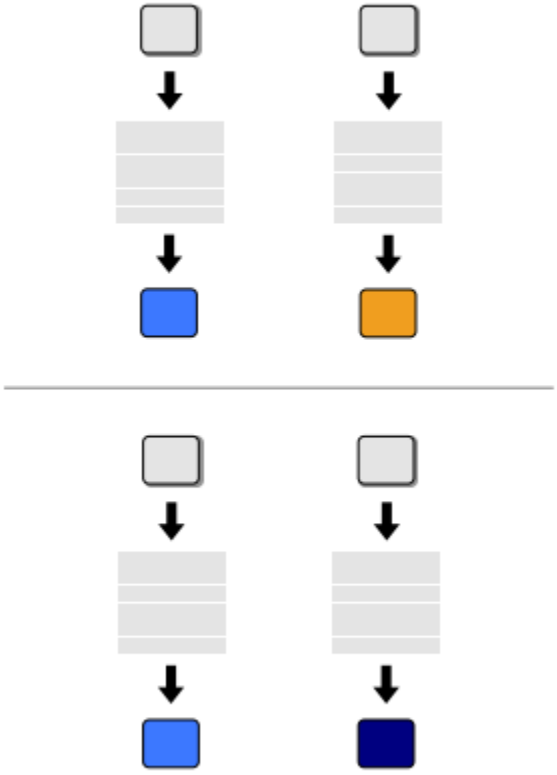
GPU

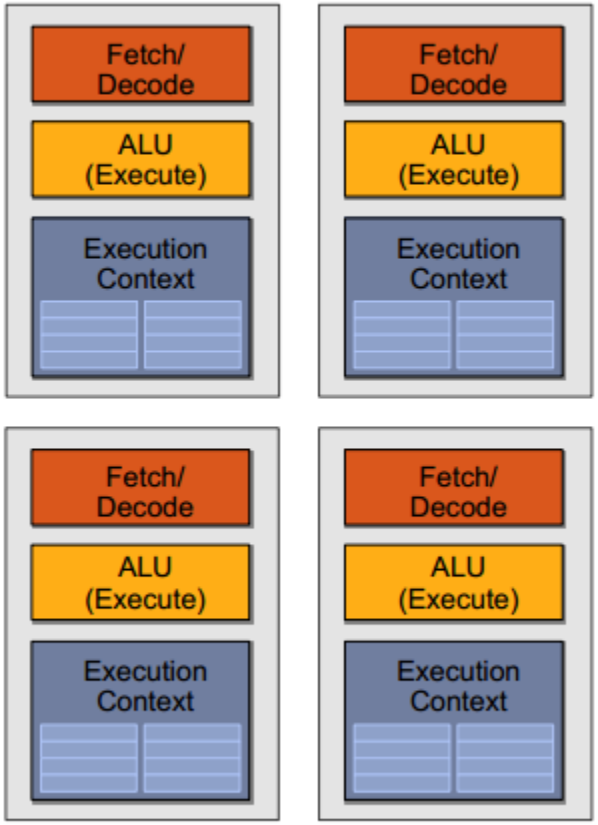
Two cores (two fragments in parallel)



GPU

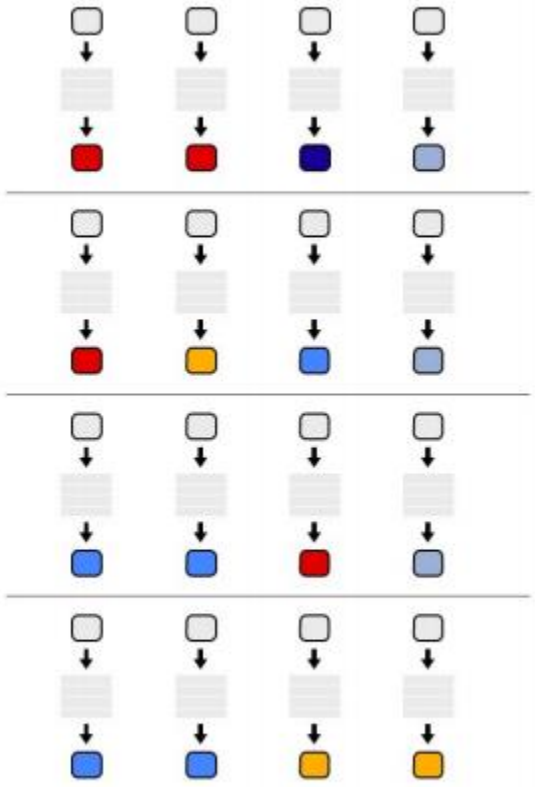
Four cores (four fragments in parallel)

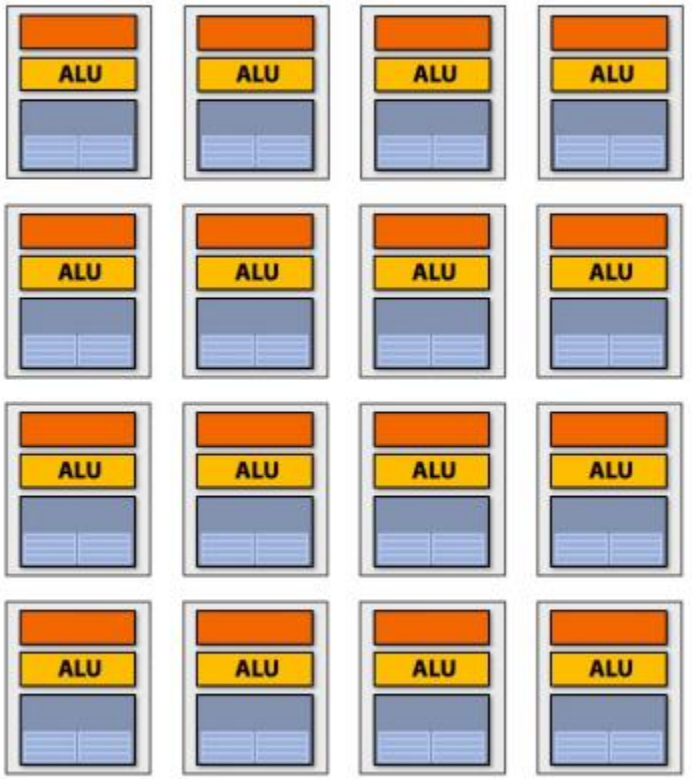




GPU

Sixteen cores (sixteen fragments in parallel)

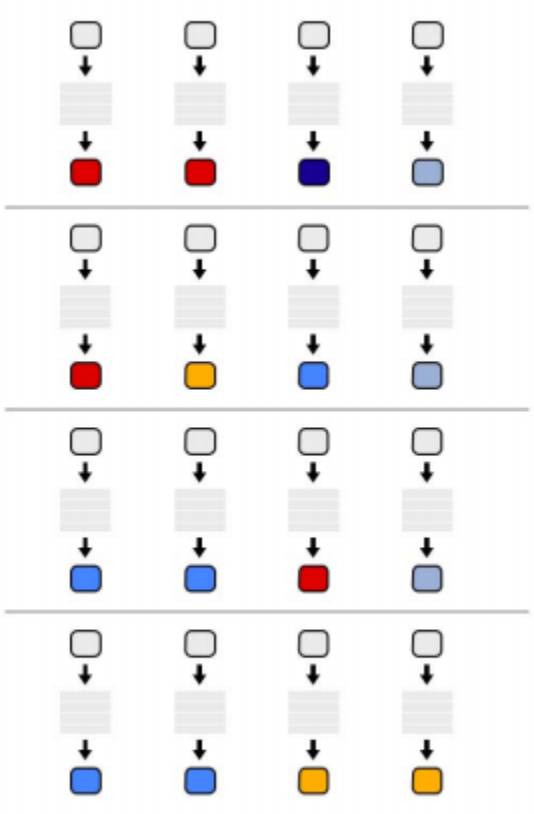




16 cores = 16 simultaneous instruction streams

GPU

Instruction stream sharing

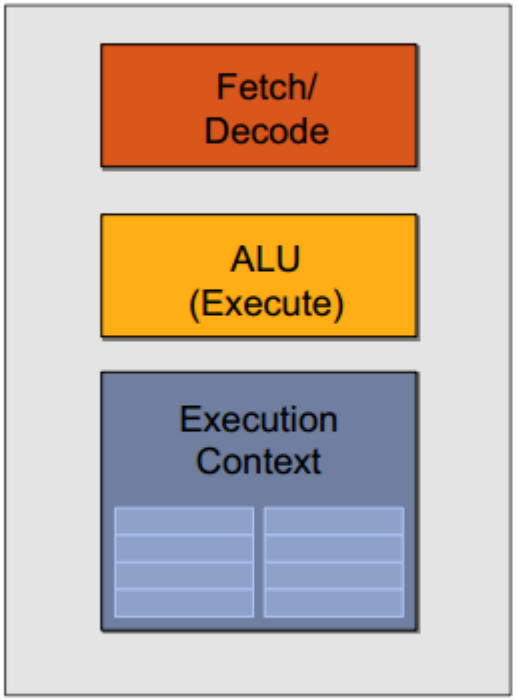


But ... many fragments should be able to share an instruction stream!

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

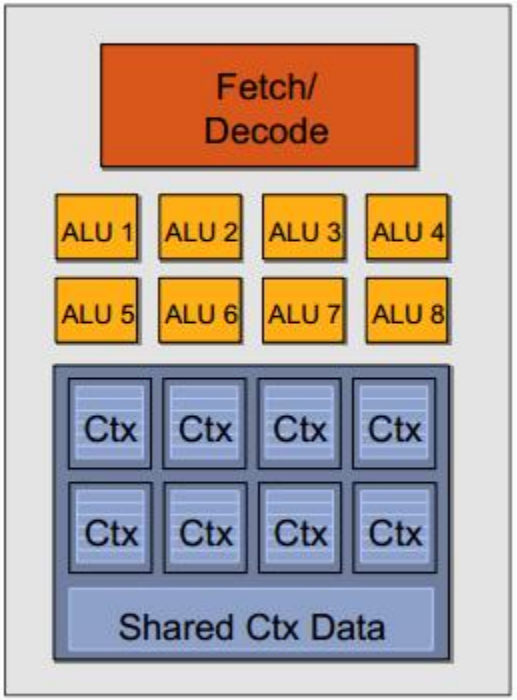
GPU

Recall: simple processing core



GPU

Add ALUs

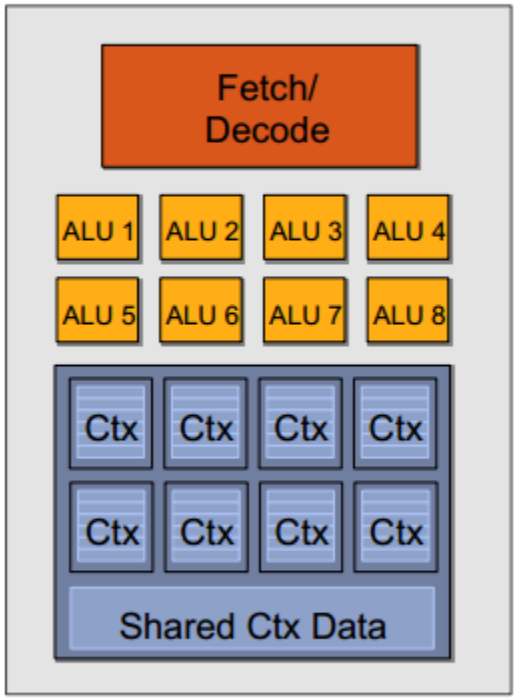


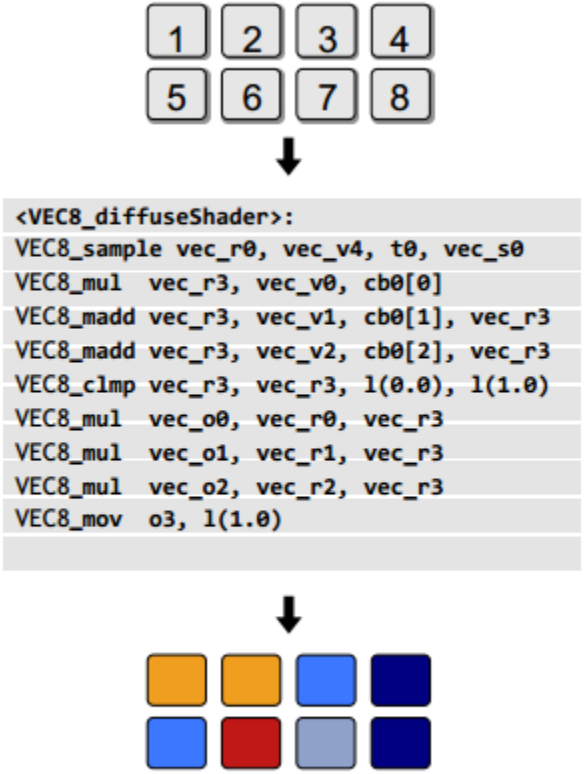
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

GPU

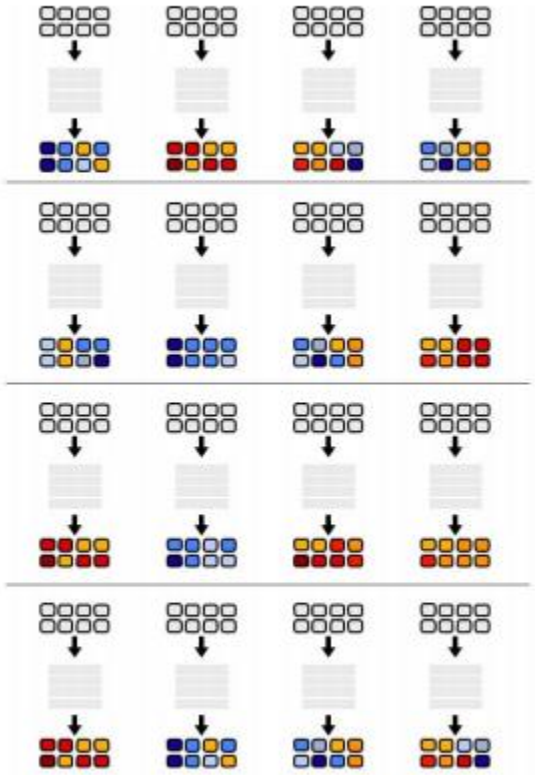
Modifying the shader

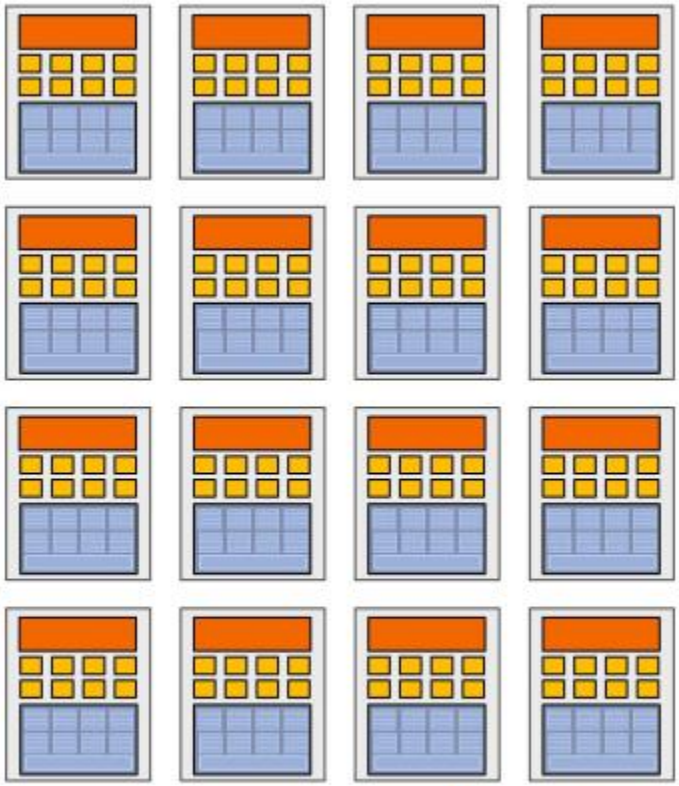




GPU

128 fragments in parallel

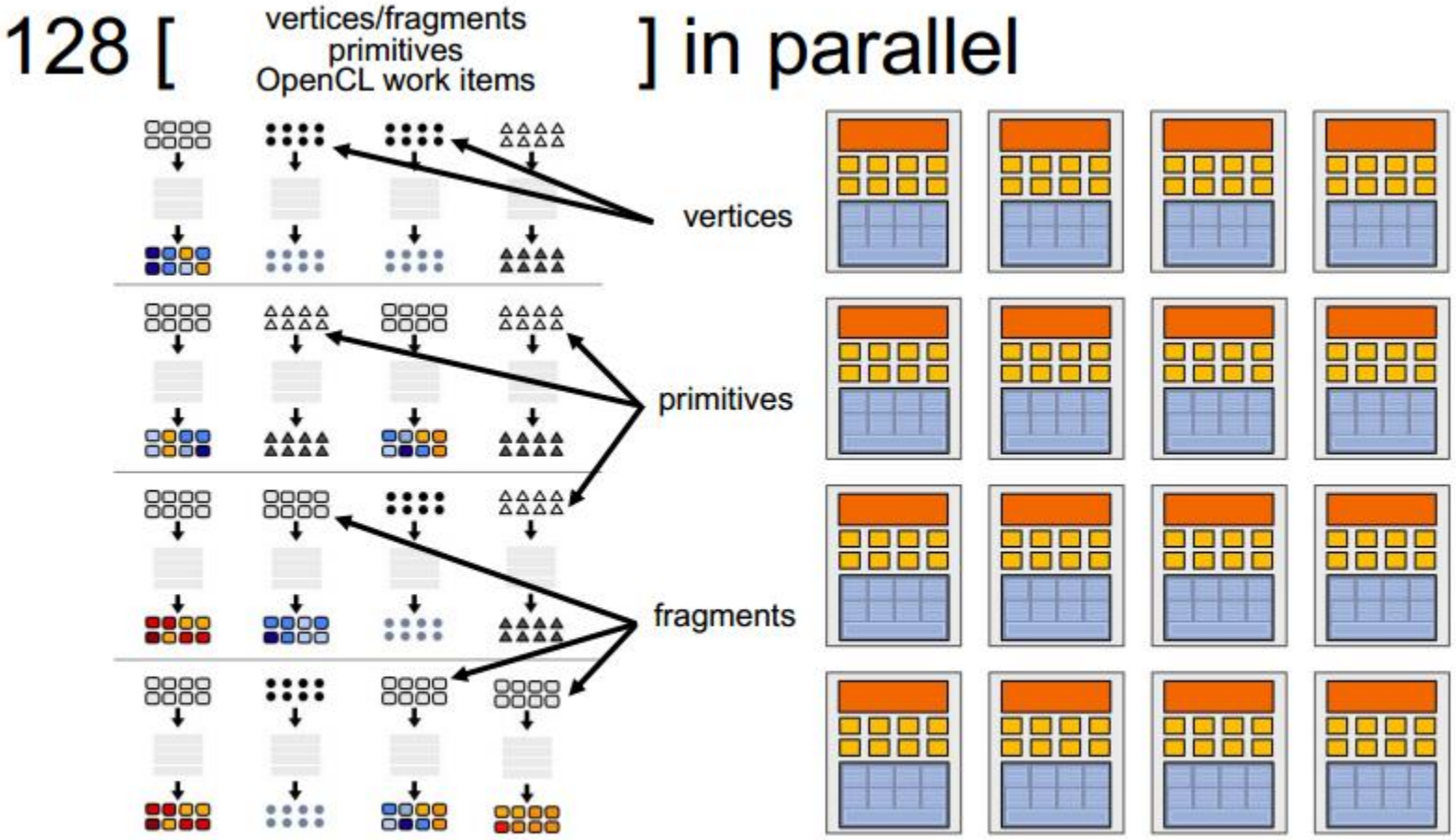




16 cores = 128 ALUs

, 16 simultaneous instruction streams

GPU


128 [
vertices/fragments
primitives
OpenCL work items
] in parallel
vertices
primitives
fragments

GPU

Clarification

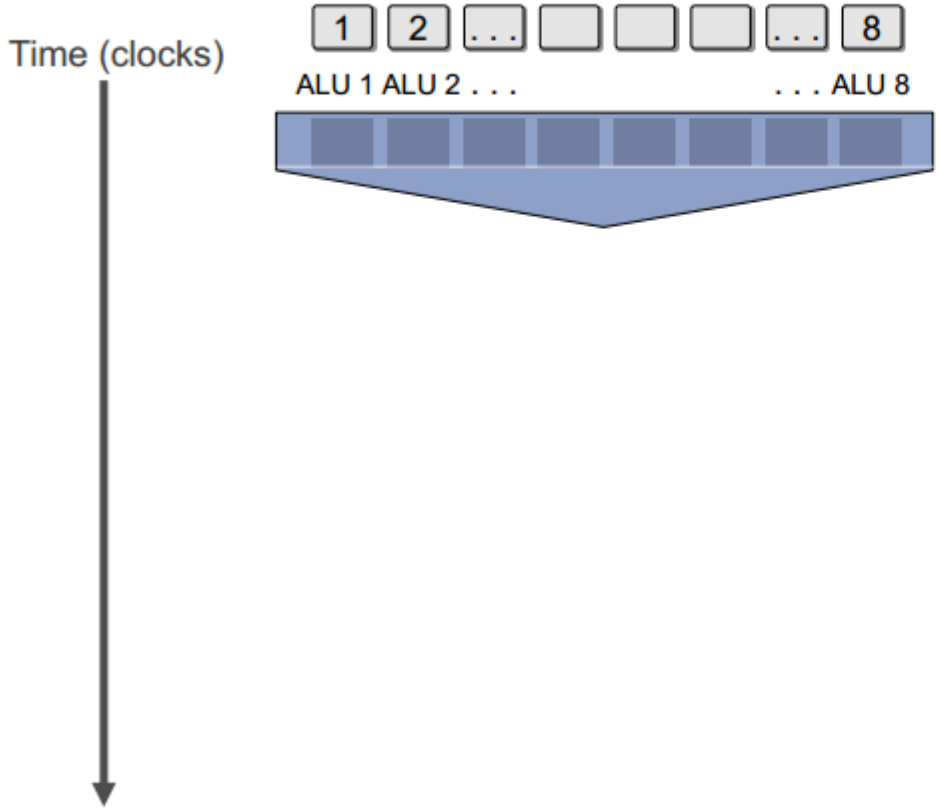
SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, AVX, Intel Larrabee
- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), ATI Radeon architectures ("wavefronts")

In practice: 16 to 64 fragments share an instruction stream.

GPU

But what about branches?

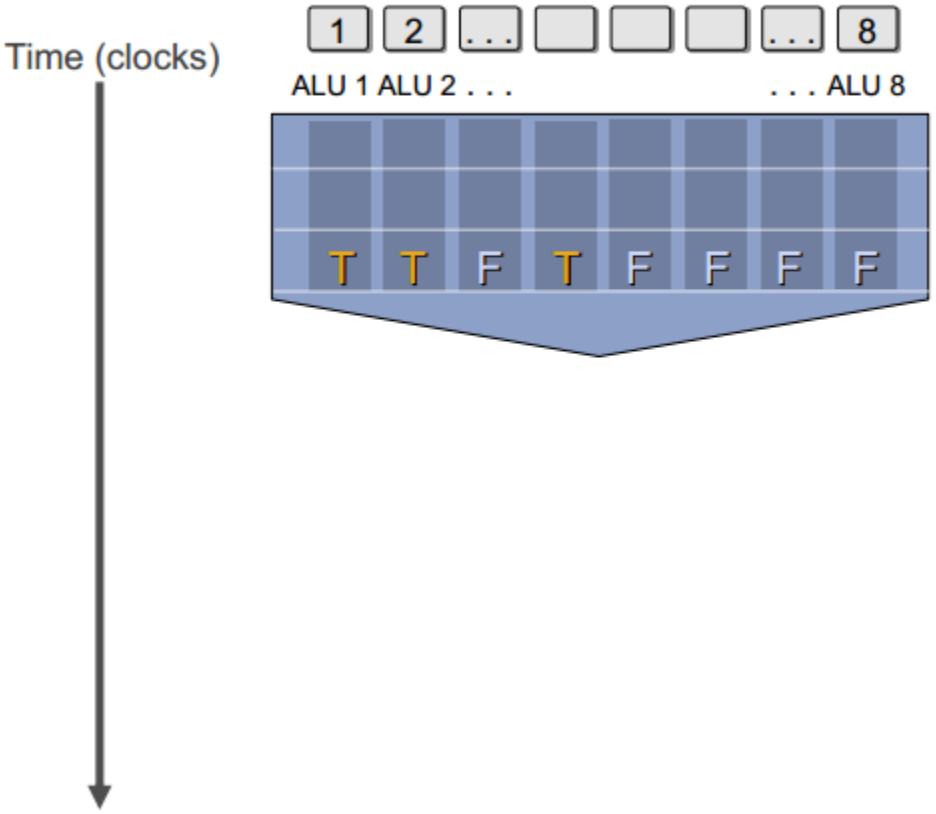



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

GPU

But what about branches?



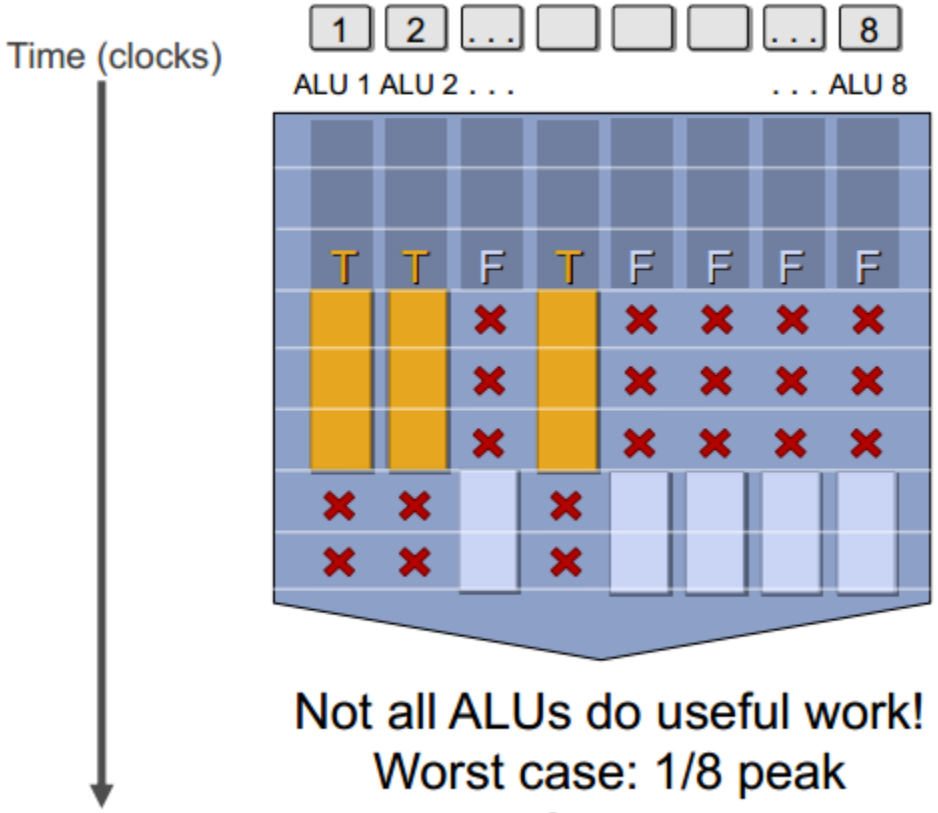

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

GPU

But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

Not all ALUs do useful work!
Worst case: 1/8 peak performance

GPU

Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

Texture access latency = 100's to 1000's of cycles

We've removed the fancy caches and logic that helps avoid stalls.

GPU

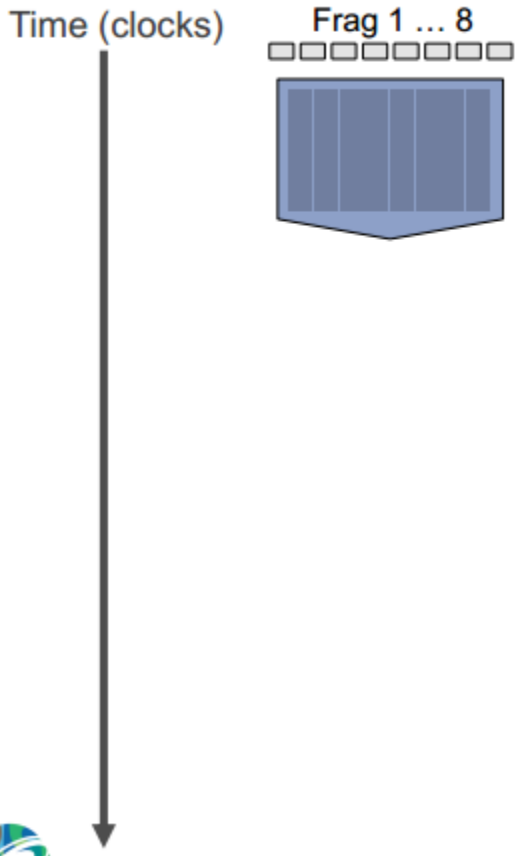
But we have LOTS of independent fragments.

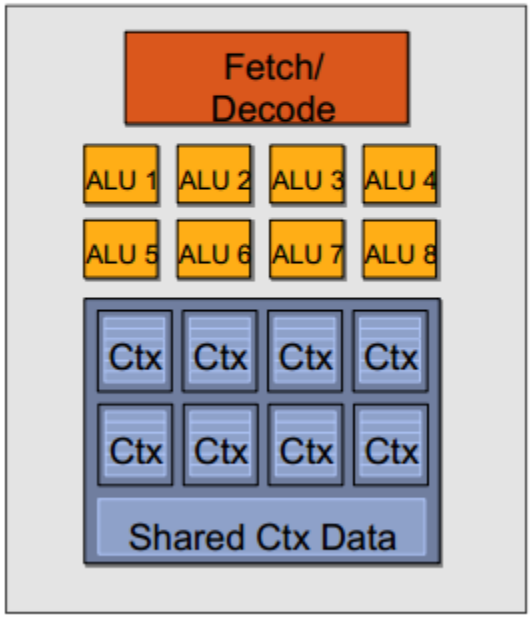
Idea #3:

Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

GPU

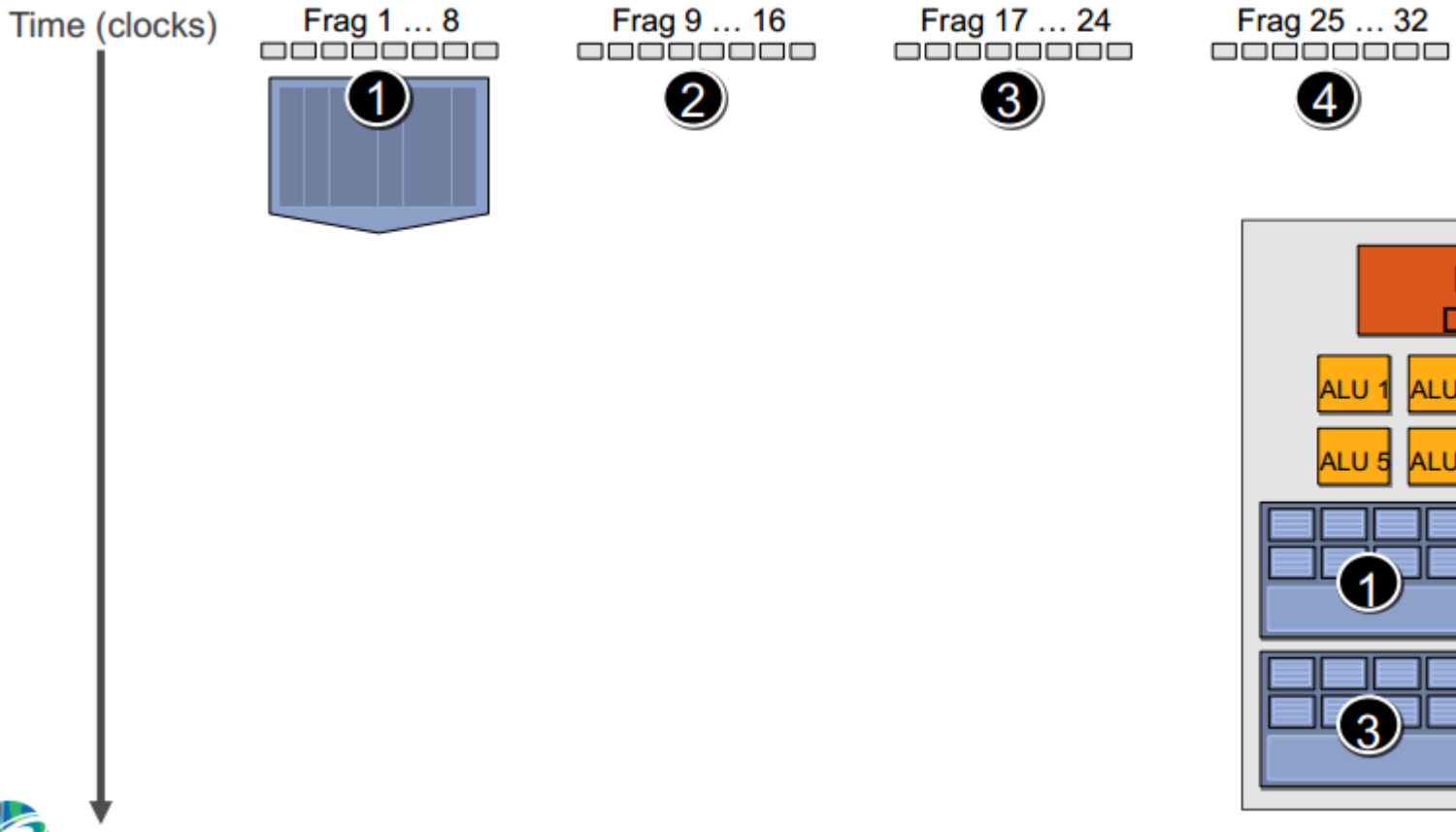
Hiding shader stalls





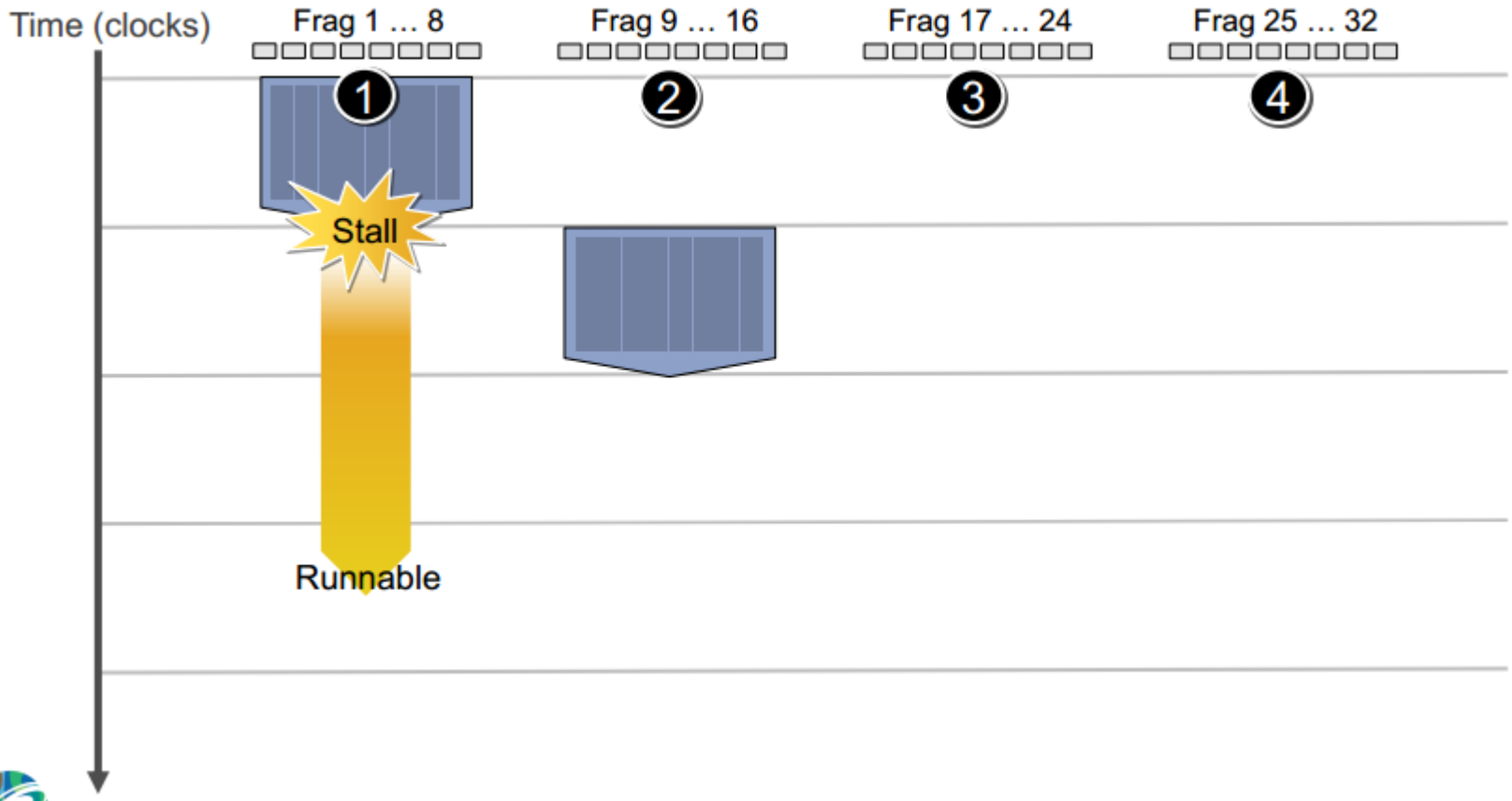
GPU

Hiding shader stalls

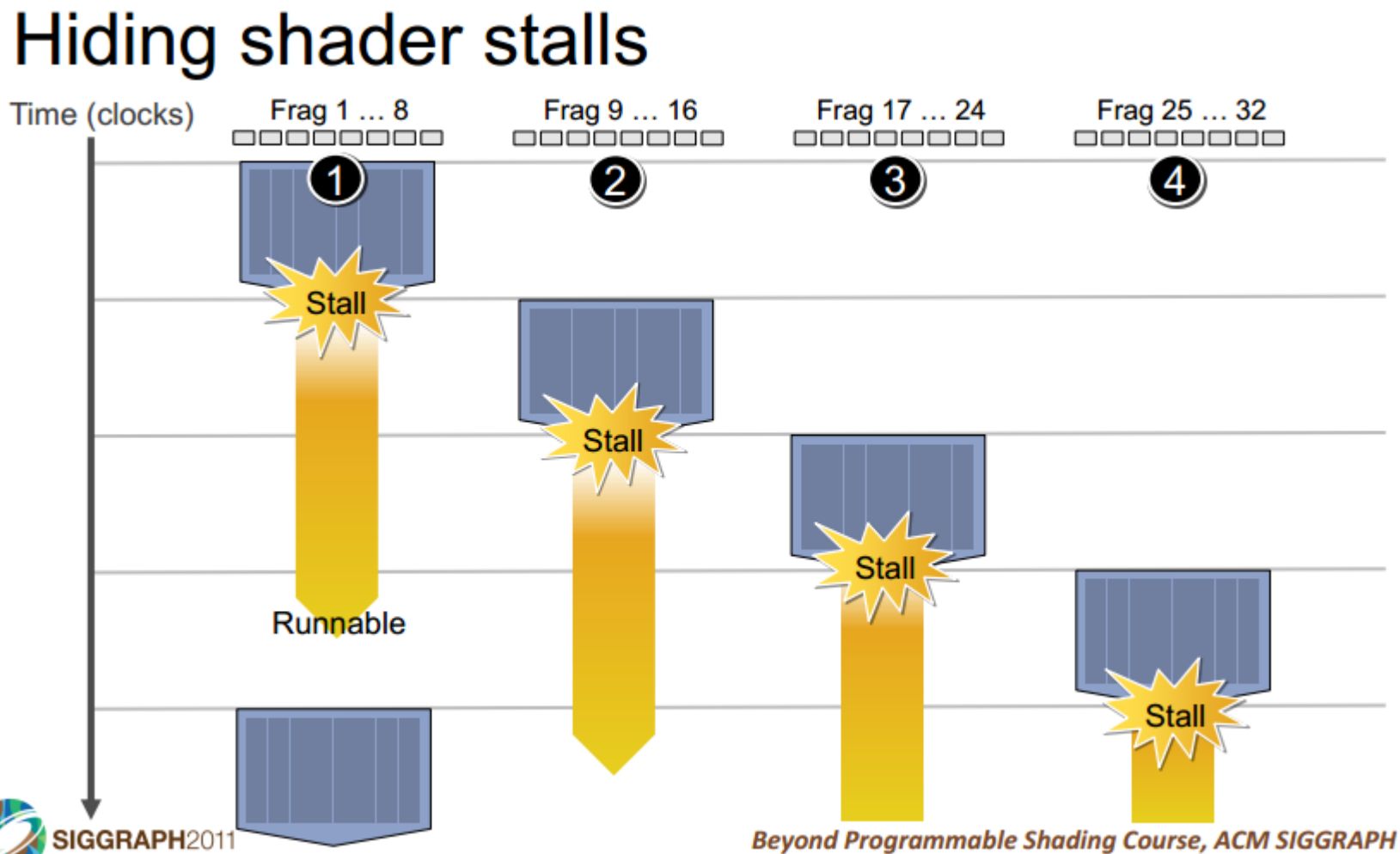


GPU

Hiding shader stalls

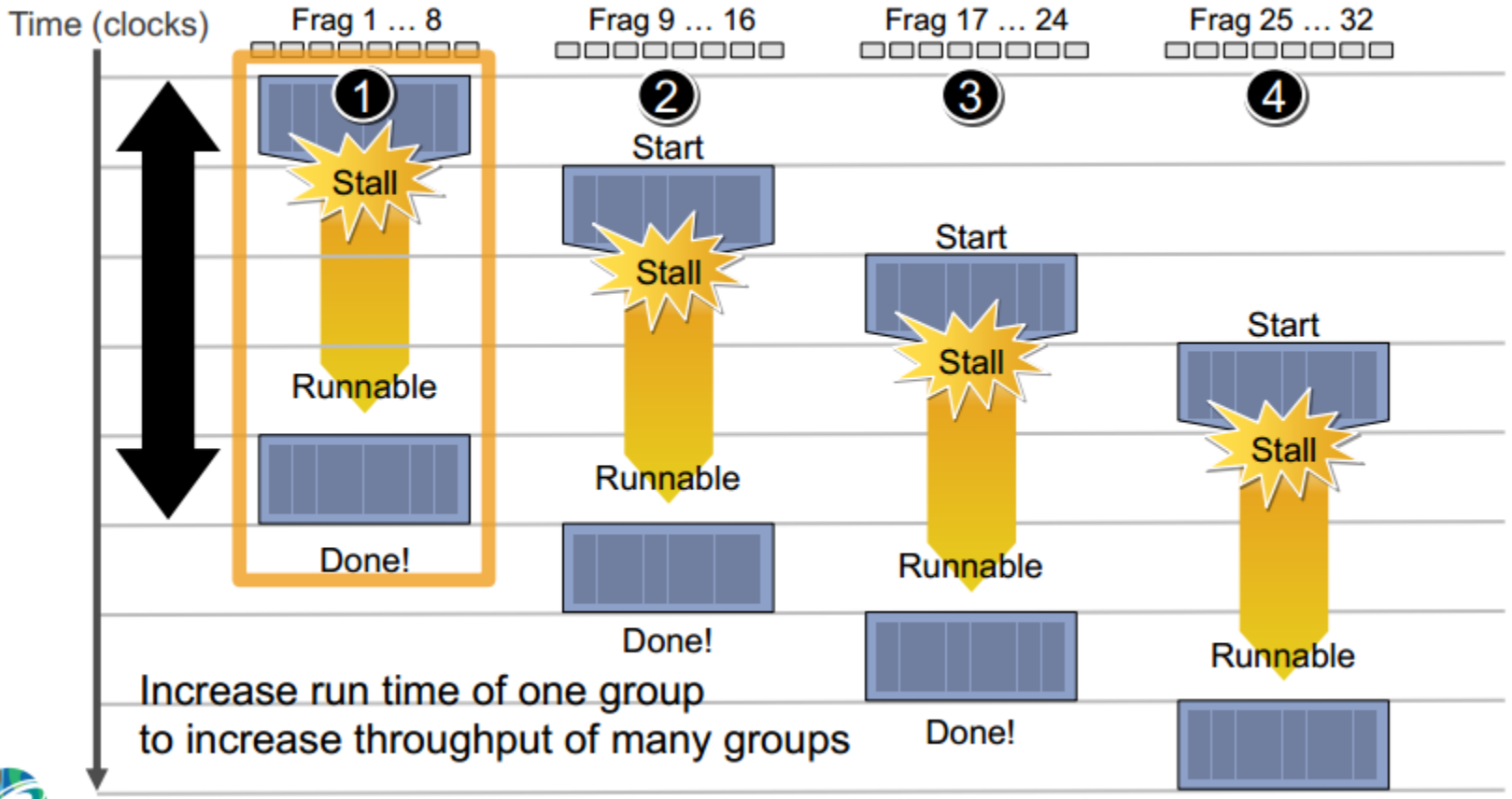


GPU


Hiding shader stalls
Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Stall
Stall
Stall
Runnable
SIGGRAPH2011
Beyond Programmable Shading Course, ACM SIGGRAPH 2011

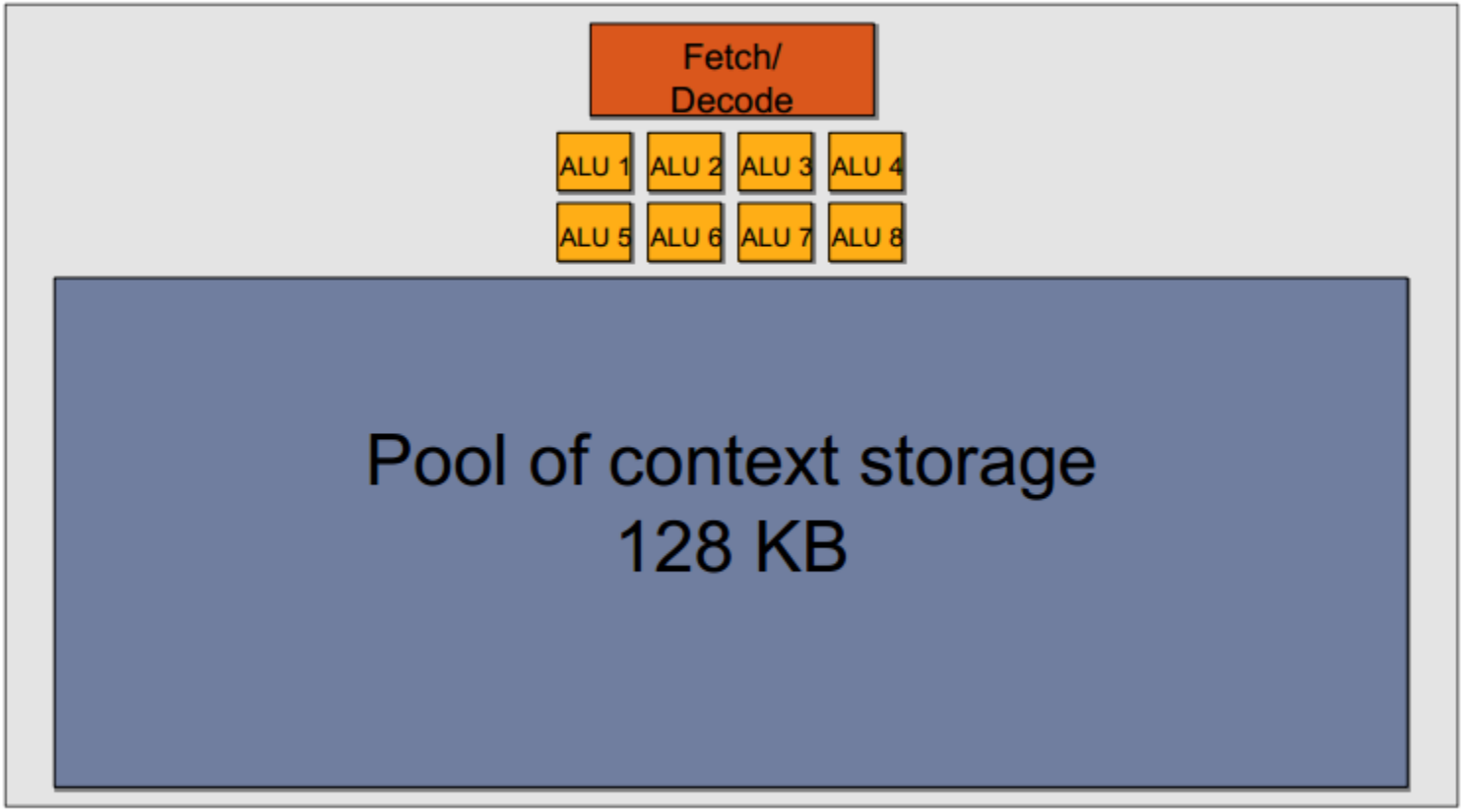
GPU

Throughput!



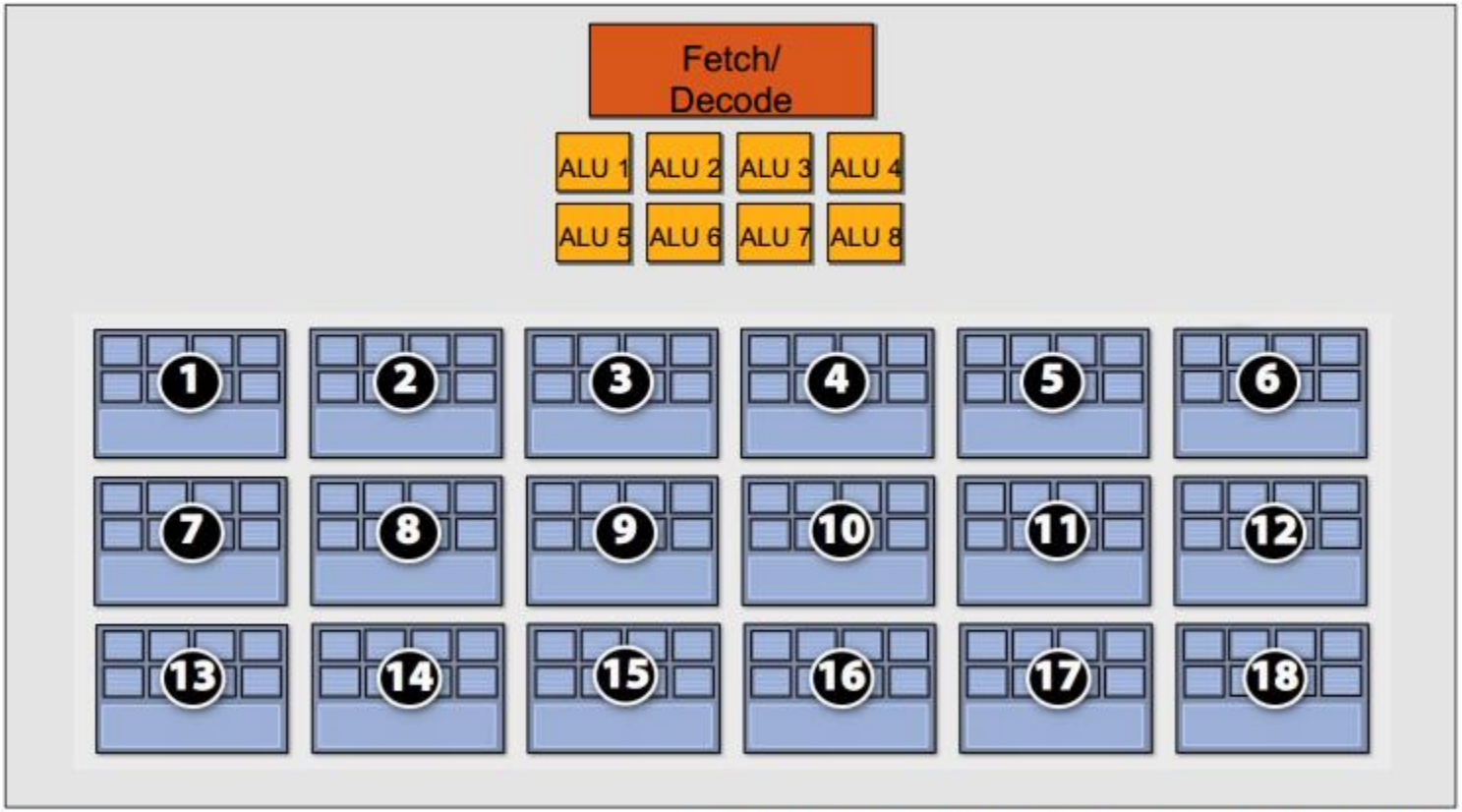
GPU

Storing contexts



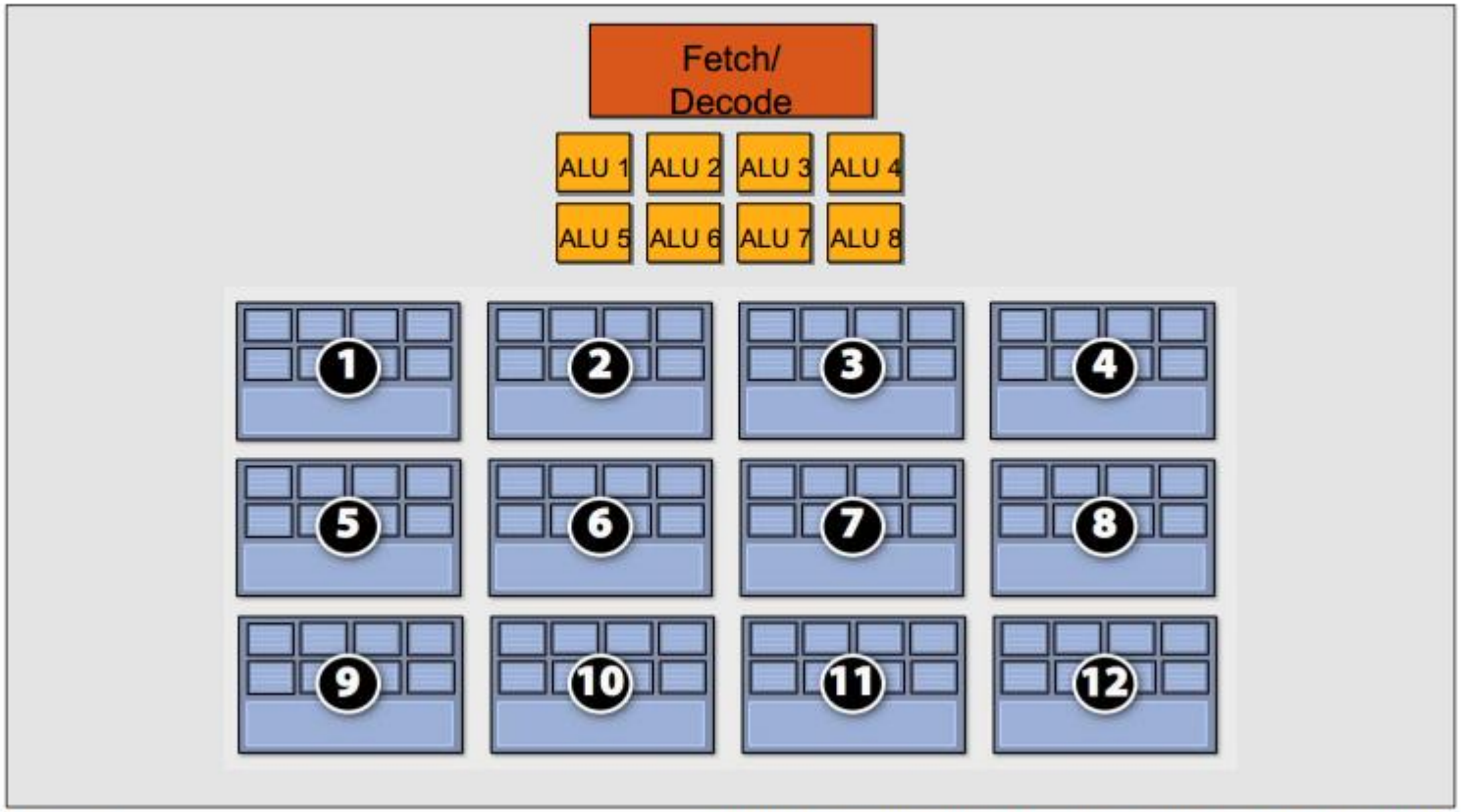
GPU

Eighteen small contexts (maximal latency hiding)



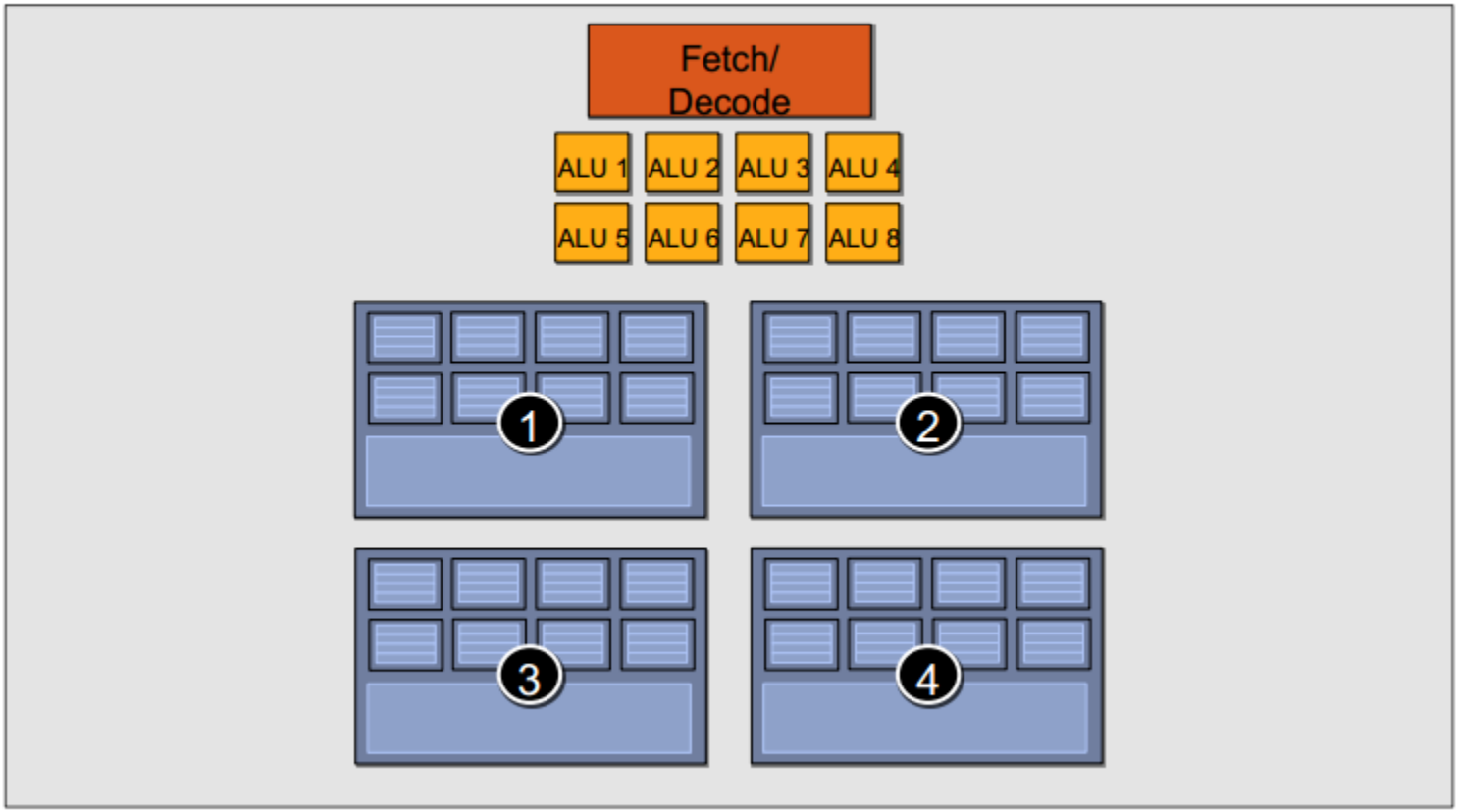
GPU

Twelve medium contexts



GPU

Four large contexts (low latency hiding ability)



GPU

Clarification

Interleaving between contexts can be managed by hardware or software (or both!)

- NVIDIA / ATI Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds fragment state
- Intel Larrabee
  - HW manages four x86 (big) contexts at fine granularity
  - SW scheduling interleaves many groups of fragments on each HW context
  - L1-L2 cache holds fragment state (as determined by SW)

GPU

Example chip

16 cores

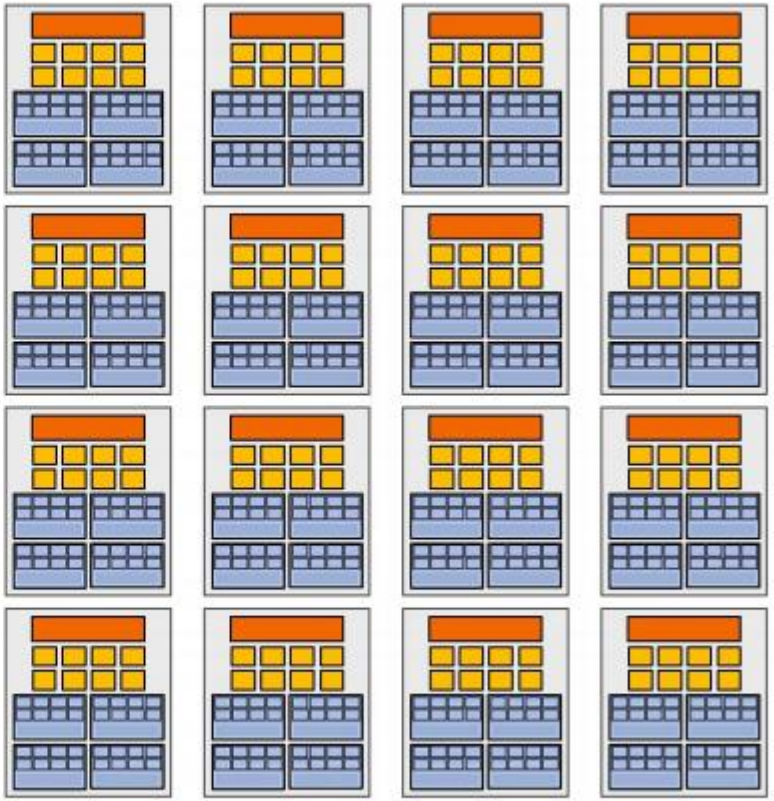
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



GPU

Summary: three key ideas

1. Use many “slimmed down cores” to run in parallel

2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

FPGA acceleration

FPGA

Computer architecture combining some of the flexibility of software with the high performance of hardware by processing with very flexible high speed computing fabrics like field-programmable gate arrays (FPGAs).

- The principal difference when compared to using ordinary microprocessors is the *ability to make substantial changes to the datapath itself in addition to the control flow*.

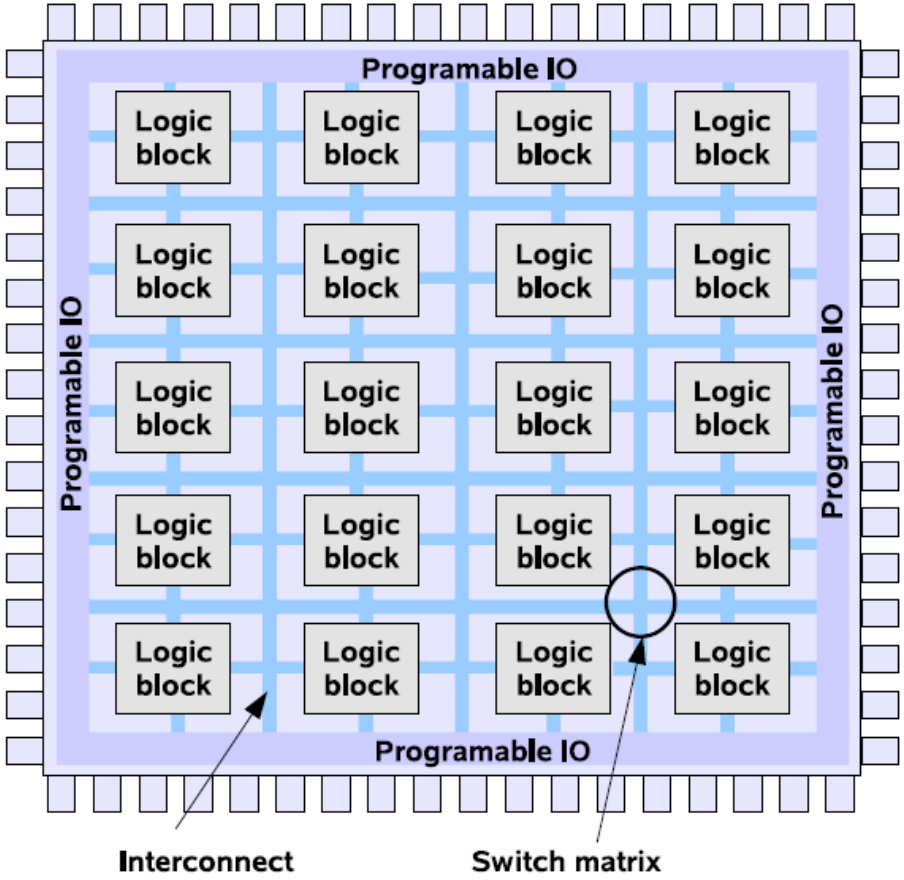
- The main difference with custom hardware, i.e. application-specific integrated circuits (ASICs) is the *possibility to adapt the hardware during runtime by "loading" a new circuit on the reconfigurable fabric.*

[wikipedia]

FPGA Architecture

The basic structure of an FPGA is composed of the following elements:

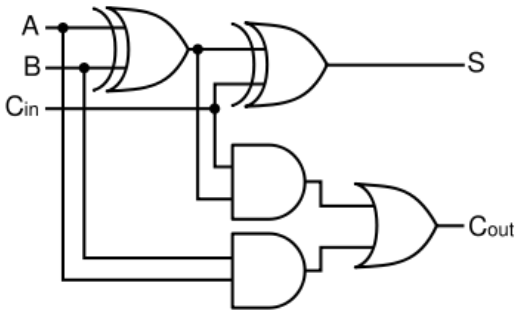
- Look-up table (LUT): This element performs logic operations
- Flip-Flop (FF): This register element stores the result of the LUT
- Wires: These elements connect elements to one another, both Logic and clock
- Input/Output (I/O) pads: These physically available ports get signals in and out of the FPGA.



FPGA Components: Logic

How can we implement any circuit in an FPGA? Combinational logic is represented by a truth table (e.g. full adder).

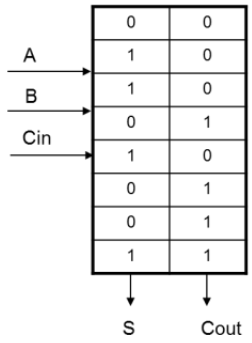
- Implement truth table in small memories (LUTs).
- A function is implemented by writing all possible values that the function can take in the LUT
- The inputs values are used to address the LUT and retrieve the value of the function corresponding to the input values



Truth Table

Inputs			Outputs	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3-input, 2-output LUT

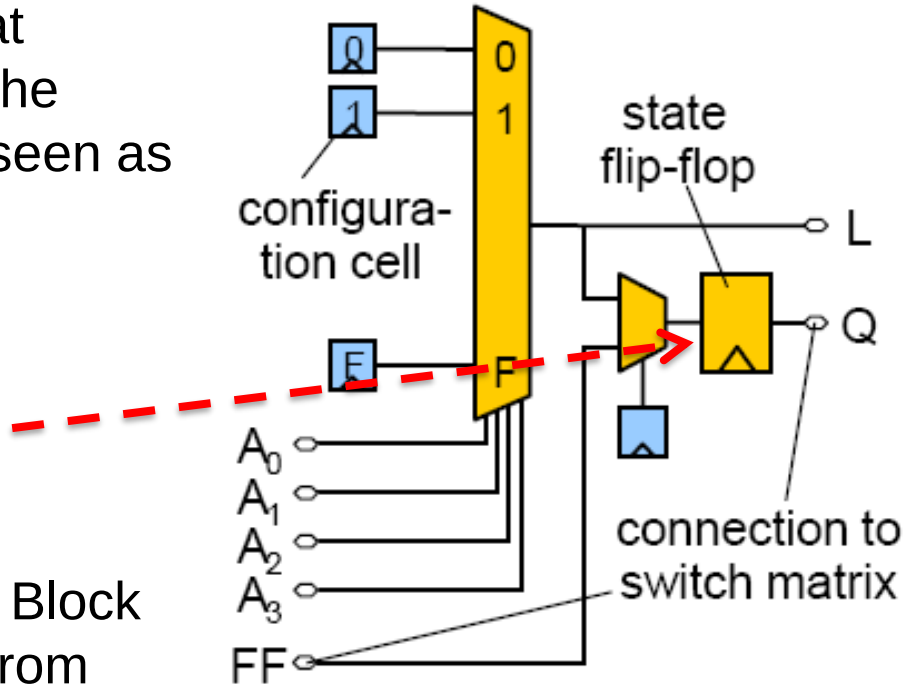


FPGA Components: Logic

A LUT is basically a multiplexer that evaluates the truth table stored in the configuration SRAM cells (can be seen as a one bit wide ROM).

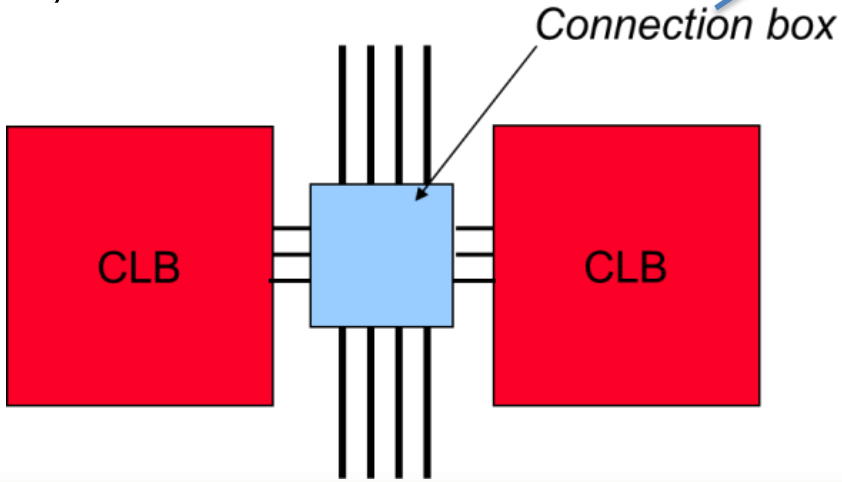
How to handle sequential logic?
Add a flip-flop to the output of LUT (Clocked Storage element).

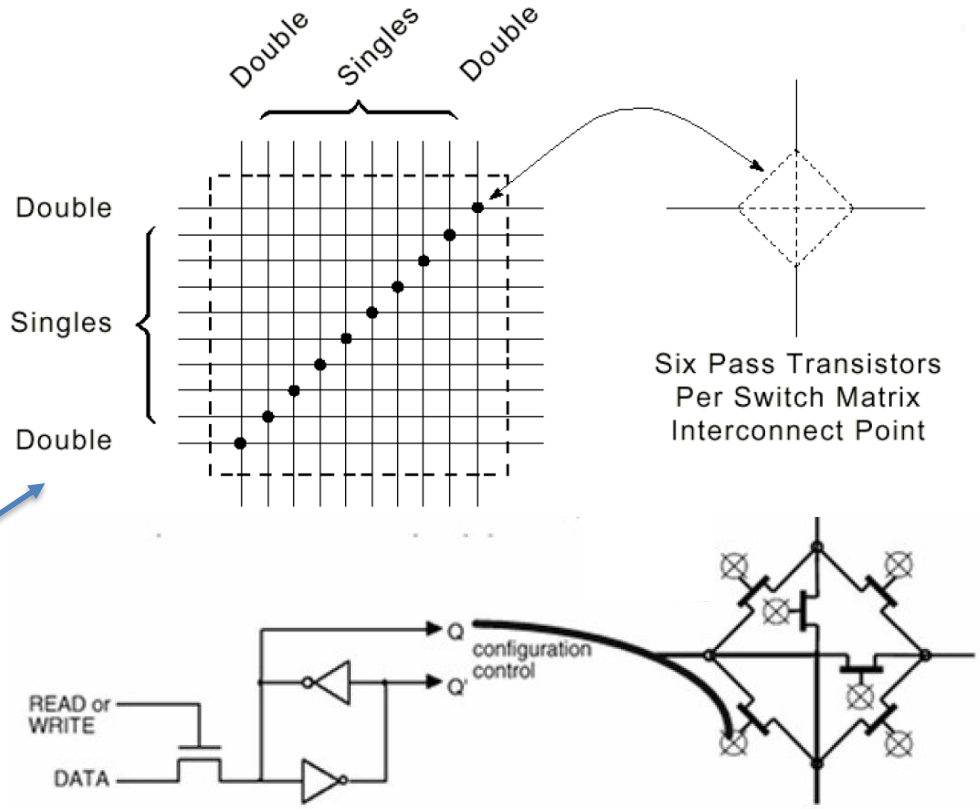
This is called a Configurable Logic Block (CLB): circuit can now use output from LUT or from FF.



FPGA Components: wires

Before FPGA is programmed, it doesn't know which CLBs will be connected: connections are design dependent, so there are wires everywhere (both for DATA and CLOCK)!!!!!
CLBs are typically arranged in a grid, with wires on all sides.





To connect CLB to wires some Connection box are used: these devices allow inputs and outputs of CLB to connect to different wires

FPGA Components: wires

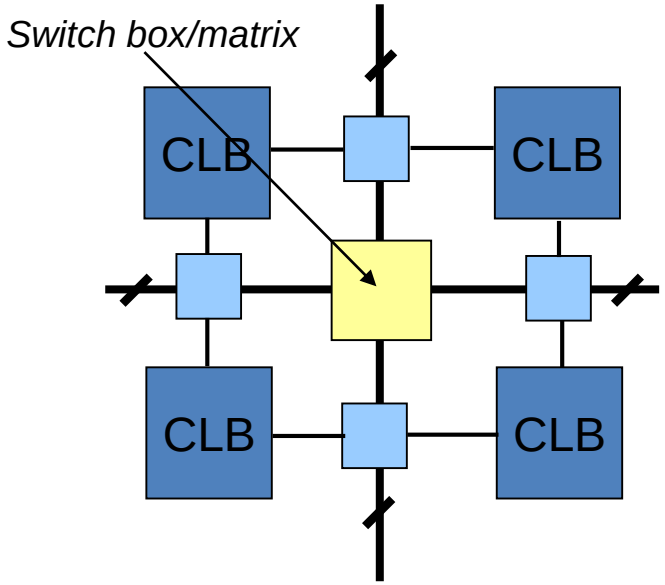
Connection boxes allow CLBs to connect to routing wires but that only allows to move signals along a single wire; to connect wires together Switch boxes (switch matrices) are used: these connect horizontal and vertical routing channels. The flexibility defines how many wires a single wire can connect into the box.

ROUTABILITY is a measure of the number of circuits that can be routed

HIGHER FLEXIBILITY

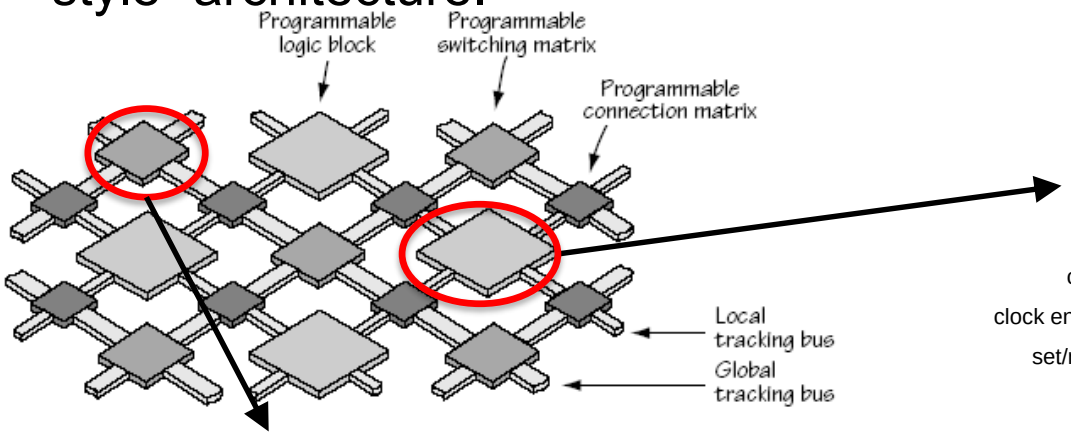
=

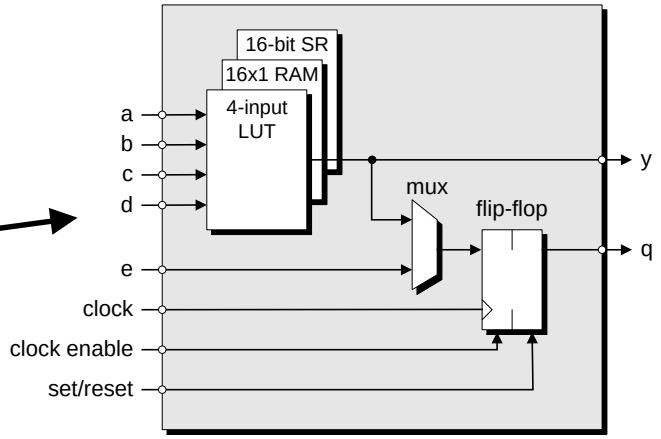
BETTER ROUTABILITY

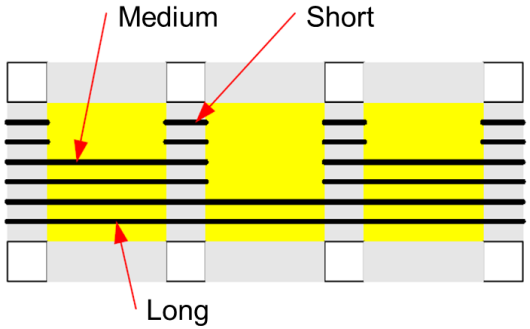


FPGA Components: wires

FPGA layout is called a “**FABRIC**”: is a 2-dimensional array of CLBs and programmable interconnections. Sometimes referred to as an “island style” architecture.

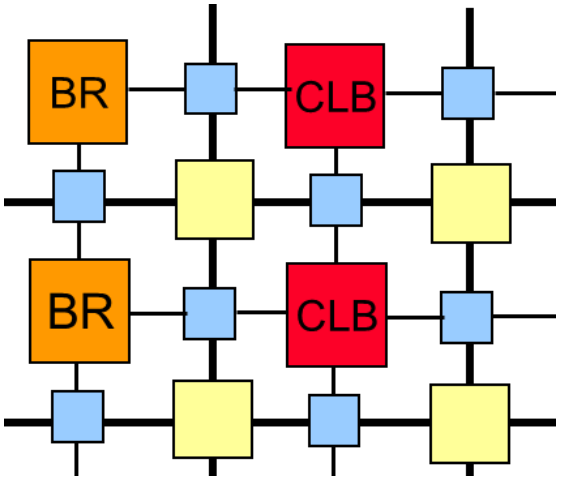






In the switch boxes there are short channels (useful for connecting adjacent CLBs) and long channels (useful for connecting CLBs that are separated, this reduce routing delay for non-adjacent CLBs)

FPGA Components: memory



The FPGA fabric includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAMs (BRAMs), LUTs, and shift registers.

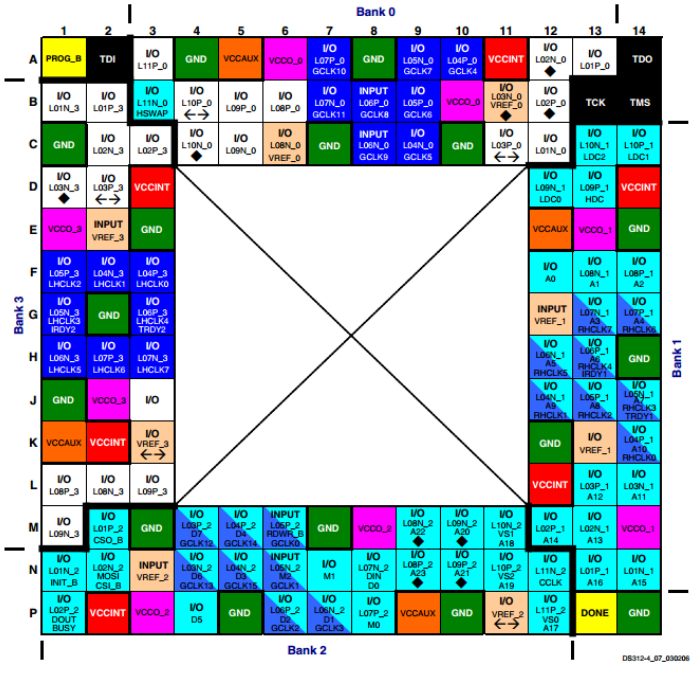
Using LUTs as SRAM, this is called

DISTRIBUTE RAM

Included dedicated RAM components in the FPGA fabric are called

BLOCKs RAM

FPGA Components: input/output



The IO PAD connect the signals from the PCB to the internal logic.

The IOB are organized in banks (depending on the technology and the producer the number of IOB per bank change).

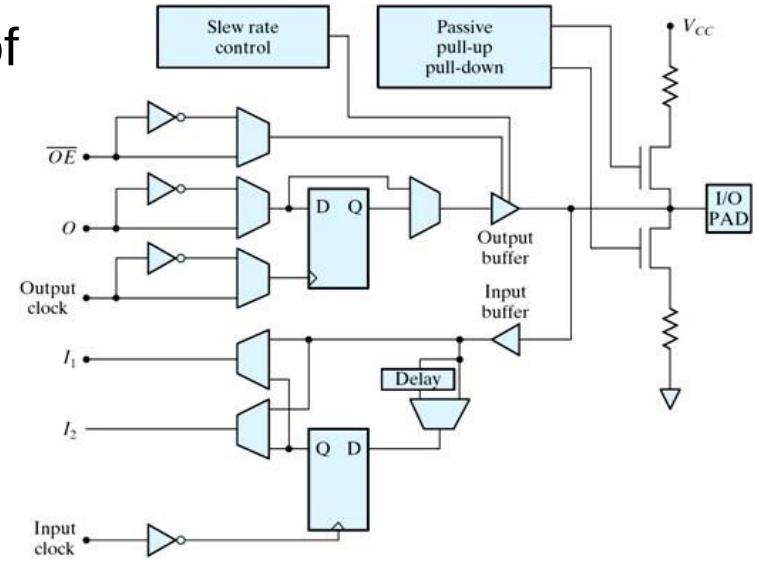
All the PAD in the same bank, share a common supply voltage: not all the different standard could be implemented at the same time in the same bank!!!!

There are special PAD for ground (GND), supplies (VCC, VCCINT, VCCAUX, etc…), clocks and for programming (JTAG).

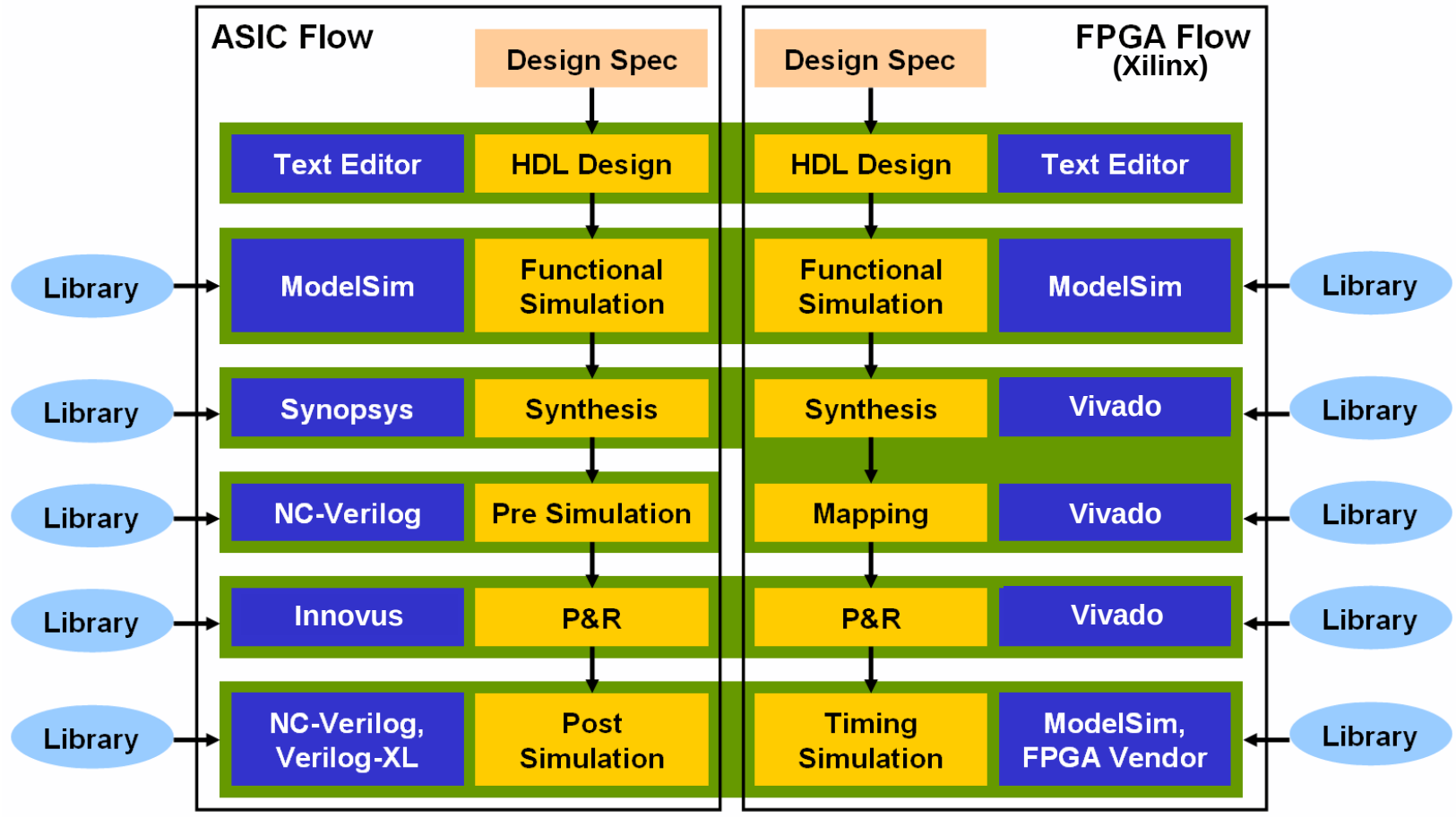
FPGA Components: input/output

The IO Blocks (IOB) support a wide range of commercial standard (LVTTL, LVCMOS, LVDS, etc…) both single ended and differential (in that case pair of contiguous pad are used).

In the PAD are available FF that are use to resynchronize the signal with the internal clock.



HW Design flow

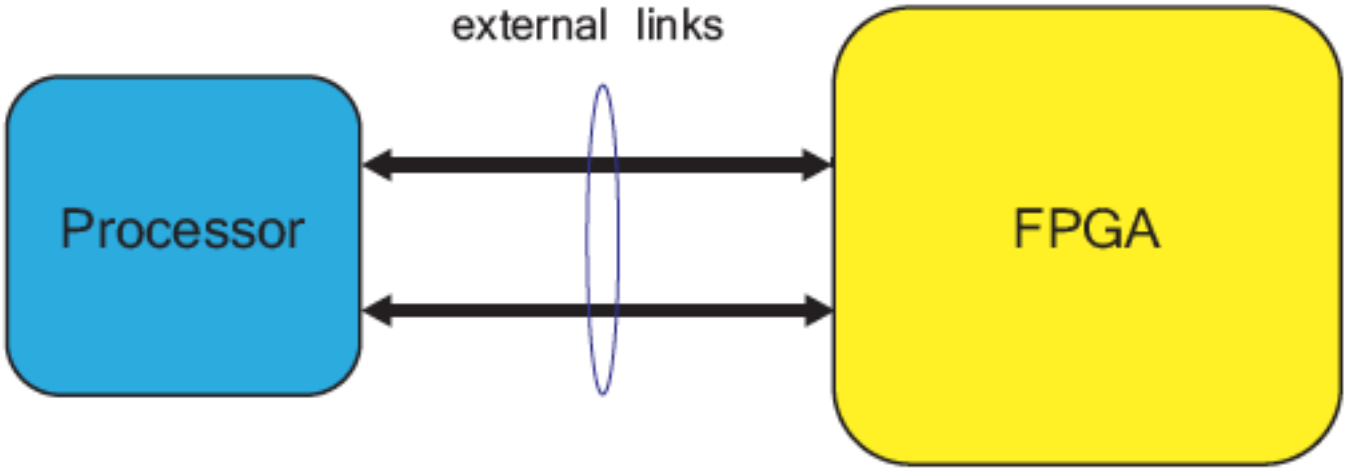


Designing with FPGA

- FPGAs are configured using a HW design flow
  - Describe the desired behavior in a HDL
  - Use the FPGA design automation tools to turn the HDL description into a configuration bitstream
- After configuration, the FPGA operates like dedicated hardware
- HW design expertise needed, low abstraction level, much slower than SW design on processors!

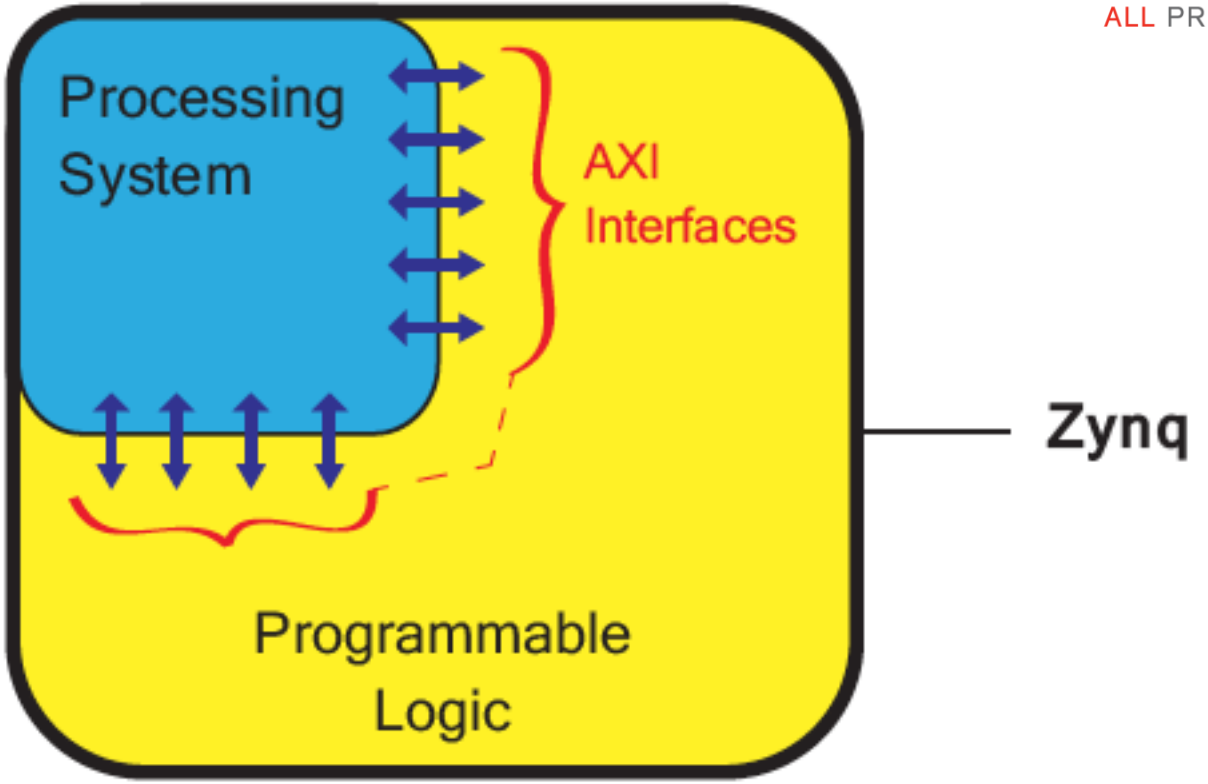
What about mixing FPGAs and Processors?

Traditional Discrete Component Architecture



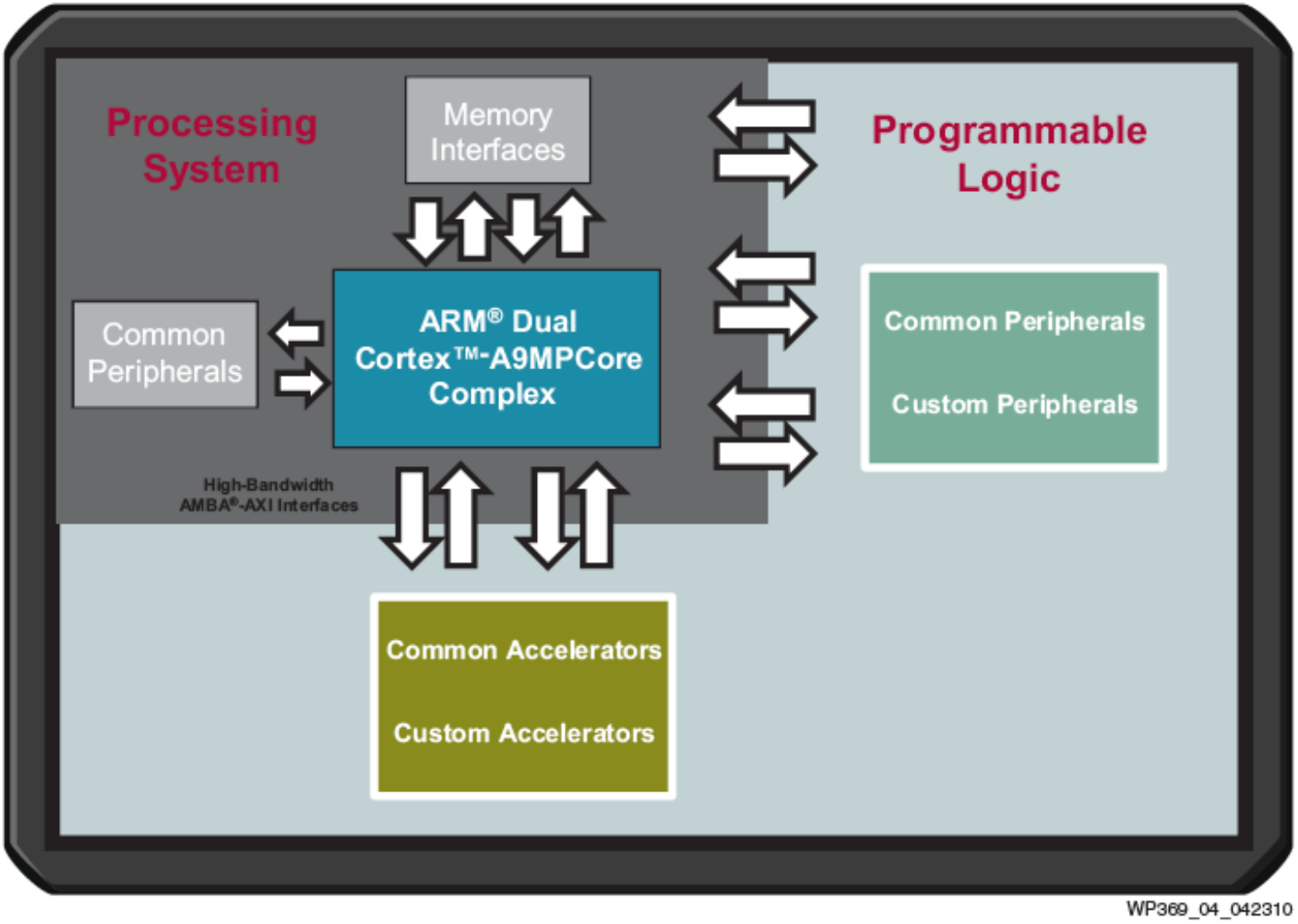
Source: The Zynq Book

Heterogenous Architecture CPU+FPGA



Source: The Zynq Book

Mapping of an Embedded SoC Hardware Architecture to Zynq



Source: Xilinx White Paper: Extensible Processing Platform

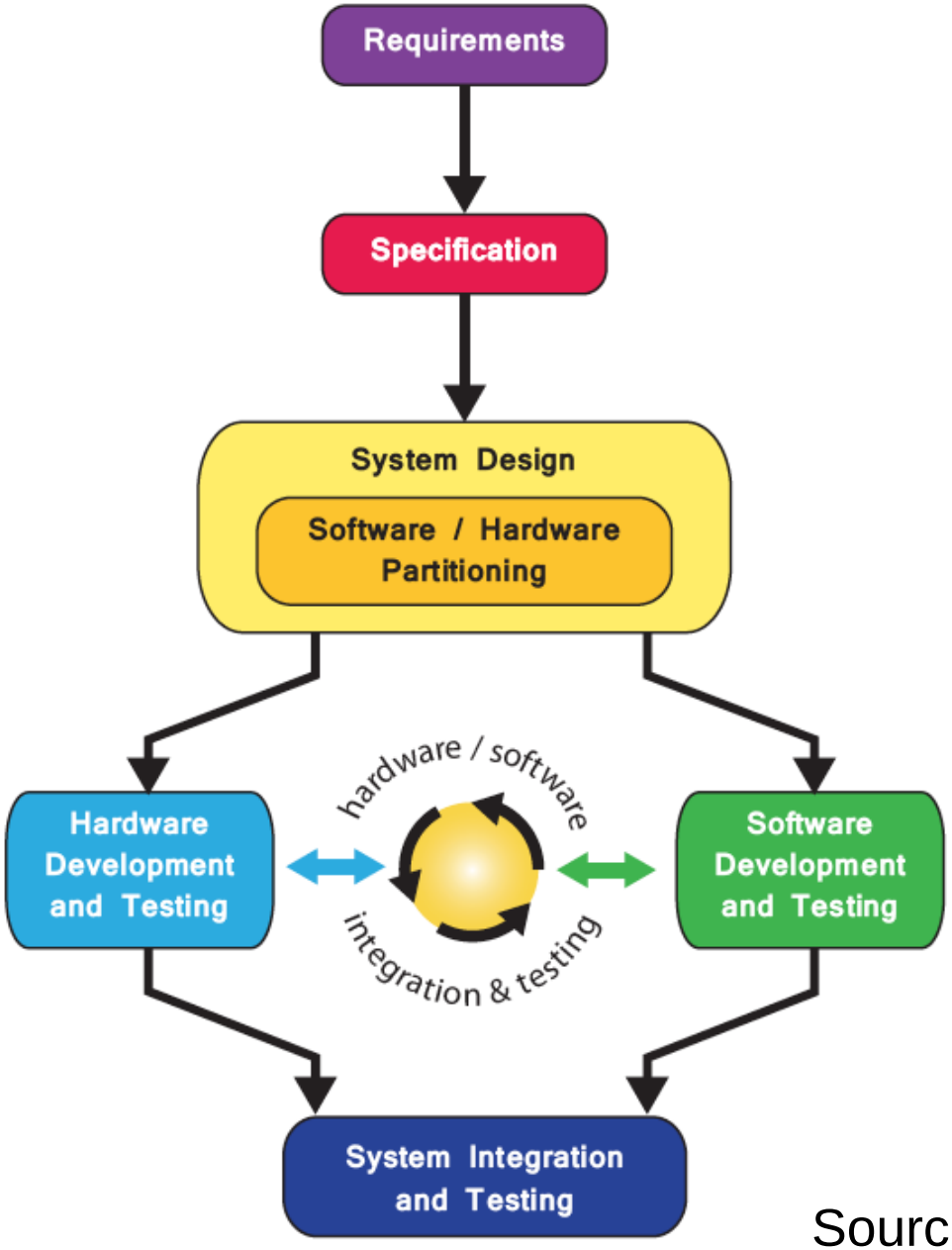
Comparison with Alternative Solutions

	ASIC	ASSP	2 Chip Solution	Zynq
Performance	+	+	■	+
Power	+	+		+
Unit Cost	+	+		■
Total Cost of Ownership	■	+	+	+
Risk		+	+	+
Time to Market		+	+	+
Flexibility			+	+
Scalability		■	+	+

positive, negative, ■ neutral

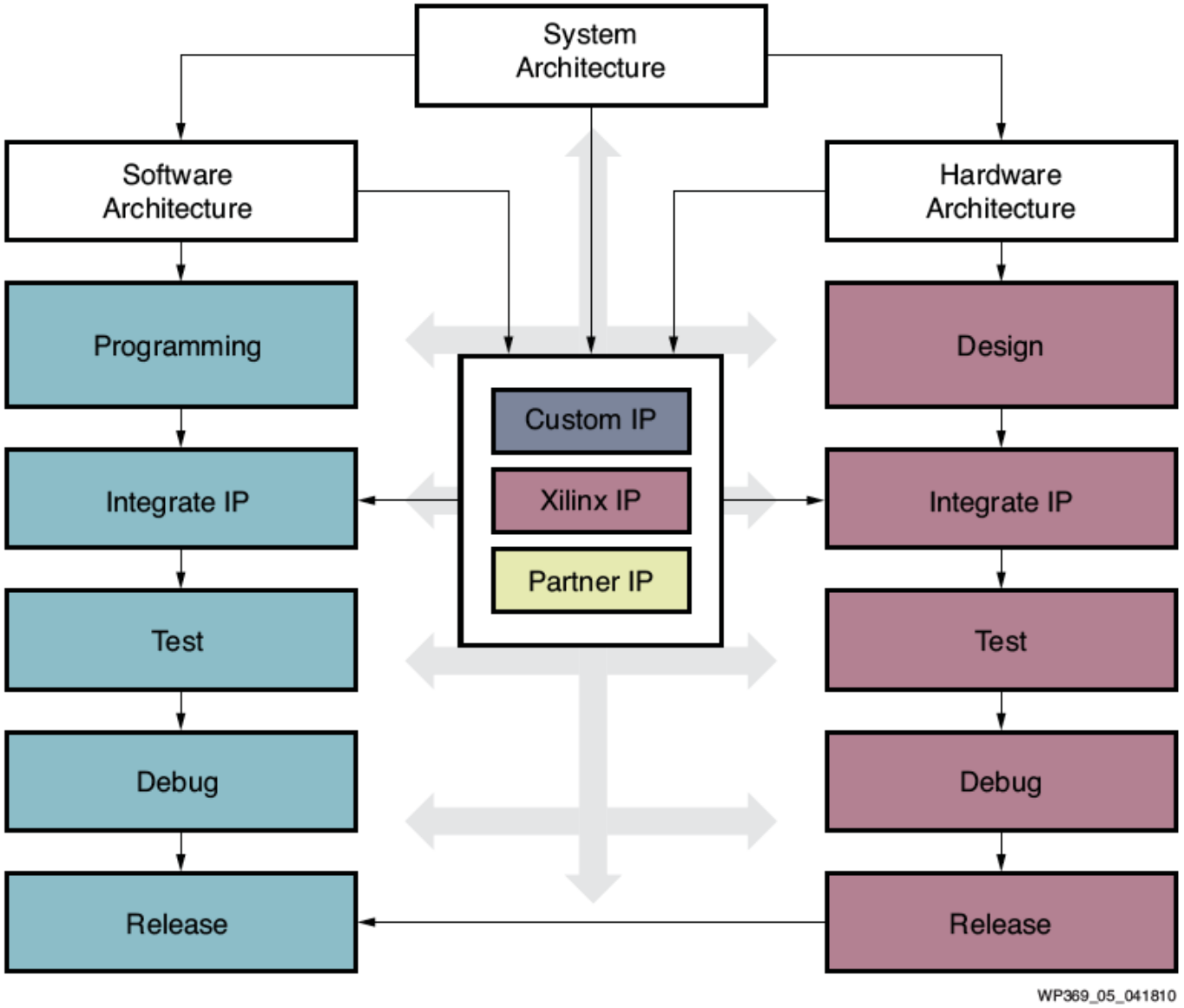
Source: Xilinx Video Tutorials

Basic Design Flow for Zynq SoC



Source: The Zynq Book

Design Flow for Zynq SoC



Source: Xilinx White Paper: Extensible Processing Platform