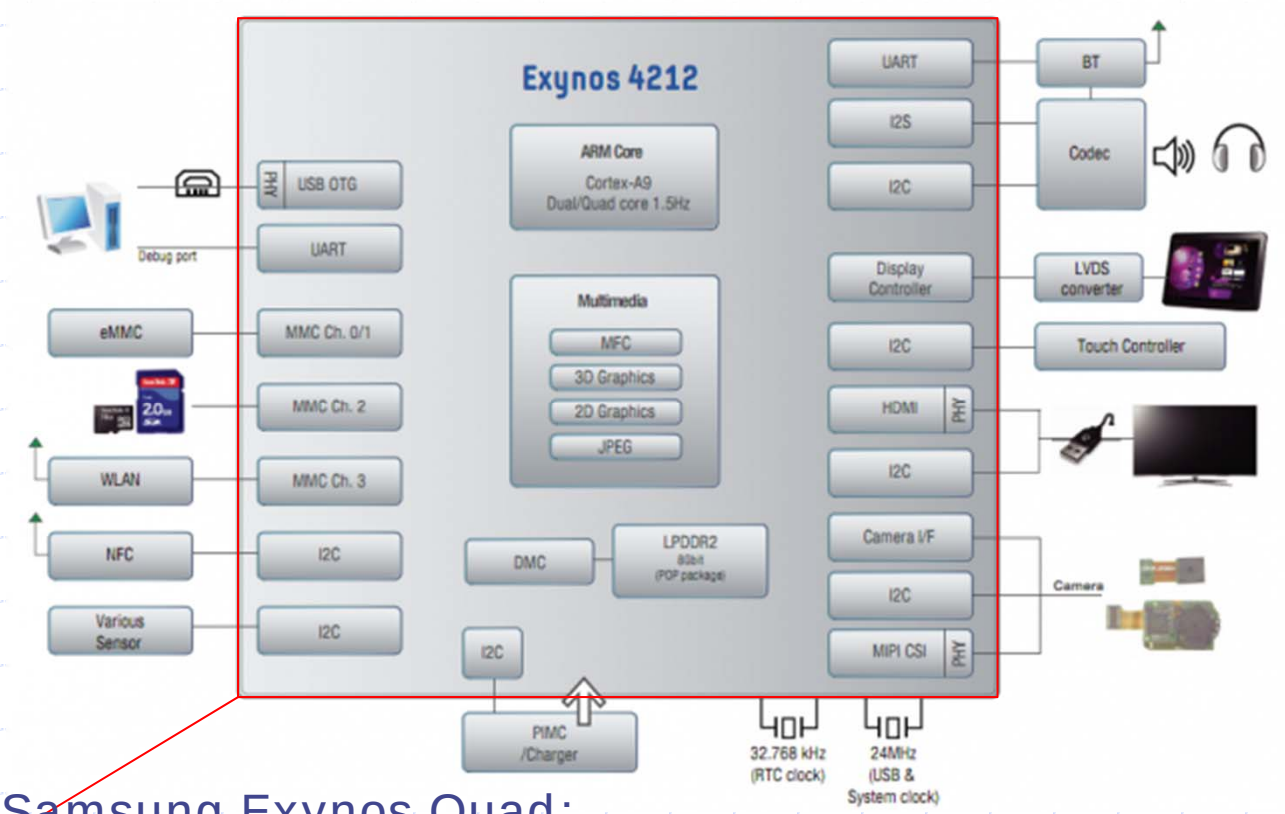
Programming in the Brave New World of Systems-on-a-chip

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

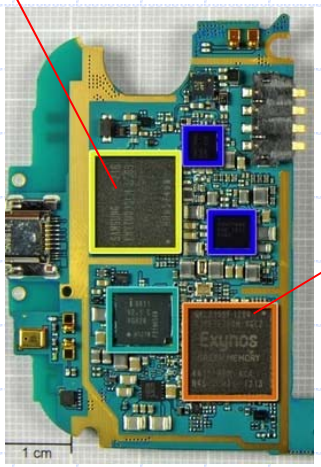
The 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)
Tokyo, Japan

September 11, 2012

Cell Phones: Samsung Galaxy S III April 2012



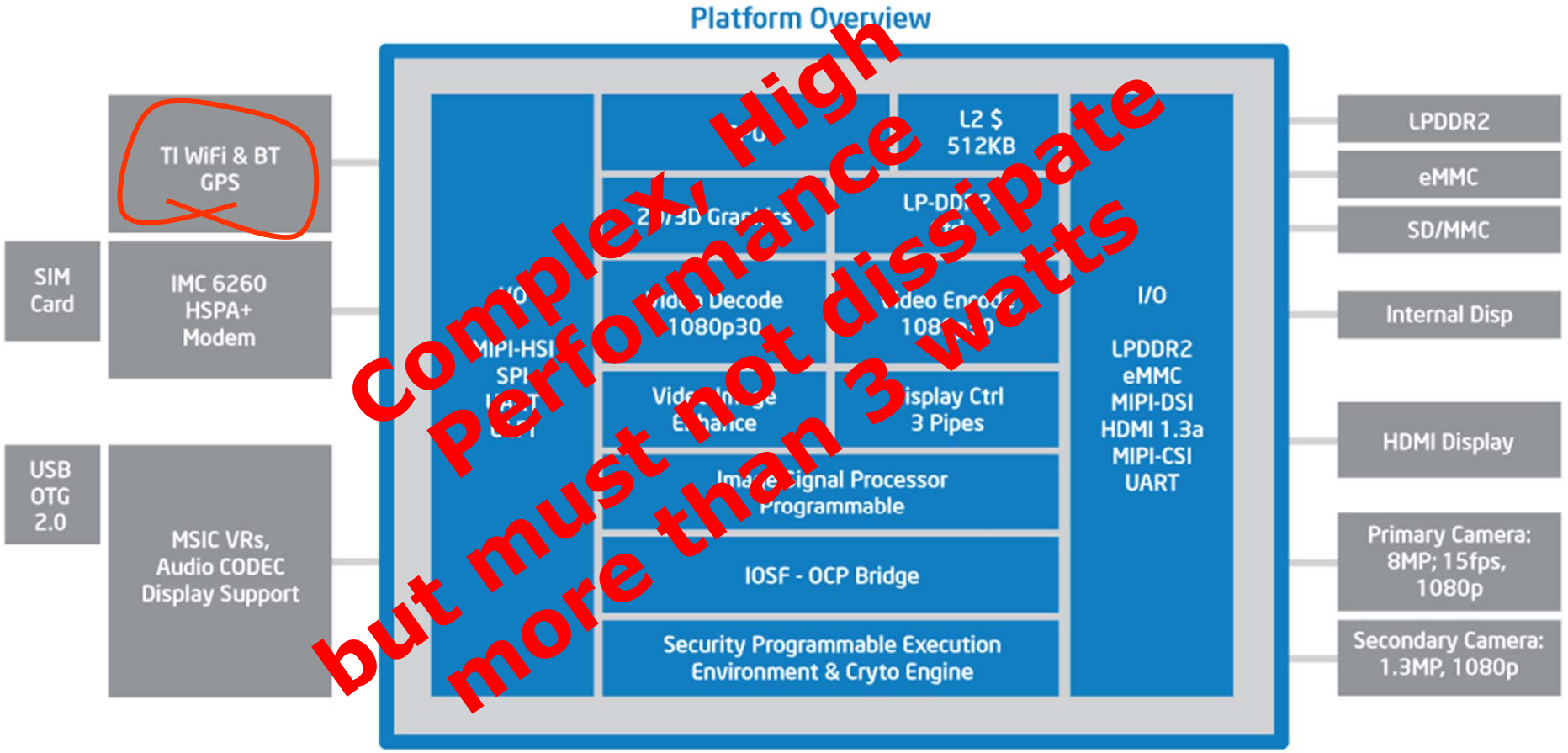
16GB
NAND flash



Samsung Exynos Quad:
- quad-core A9
- 1GB DDR2 (low power)
- Multimedia processor
- ...

power consumption <1mW

Cell Phones: Intel's Penwell Announcement January2012



Many specialized complex blocks

750 mWatt

Where does the power go

- Display
  - turn it on as little as possible
- Radios
  - Many different radios – GSM, Wimax, … Only some are turned on simultaneously
  - Energy consumption increases with data-rates, distance, moving radios, occlusions, ..
- Computation/Application
  - power consumption varies a lot from application to application
  - power hogs: H.264, Graphics, …

Power/Energy issues are the main driver in design: e.g. H.264

- Computation intensive
  - Complex predictions, IMDCT Conversions, Image smoothing
- To fit in a 3 W power budget, it has to be run mostly in specialized hardware
  - 100X power advantage over software
- A good Implementation requires design exploration
- Reusable IP requires *parameterizations to support various frame rates and sizes*

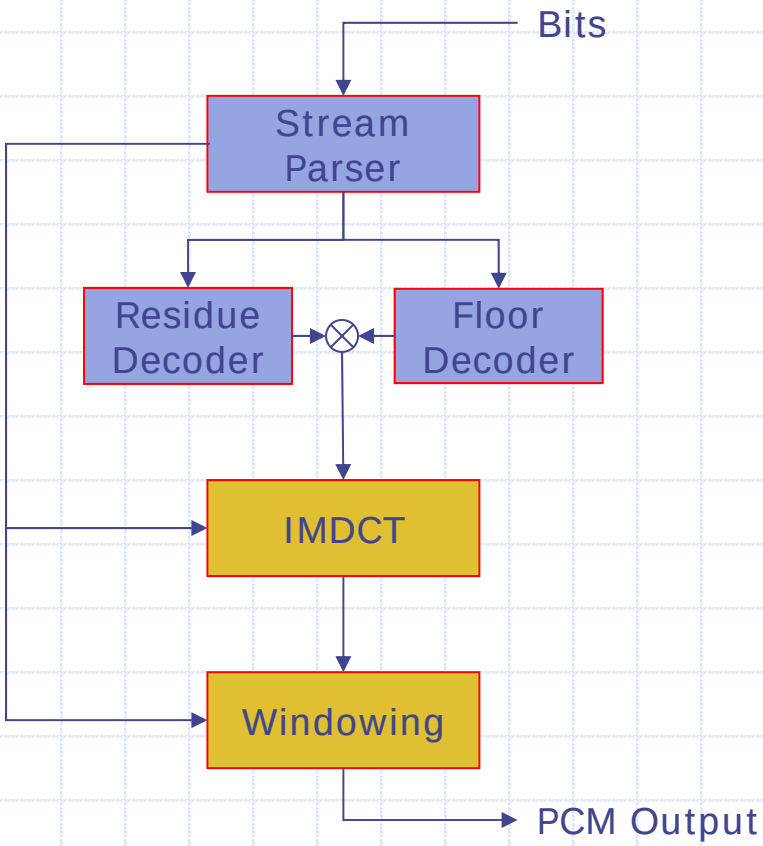
Modern SoCs: Power savings require special purpose hardware



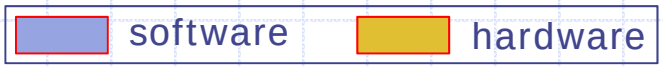
- Most SoCs employ HW acceleration for important compute intensive applications
- Software stacks on top of special purpose HW
  - Efficient interaction is a first-order concern
- Implementing algorithms to use both HW and SW is challenging – exploring many design alternatives is practically impossible

Why is exploring design alternatives difficult?

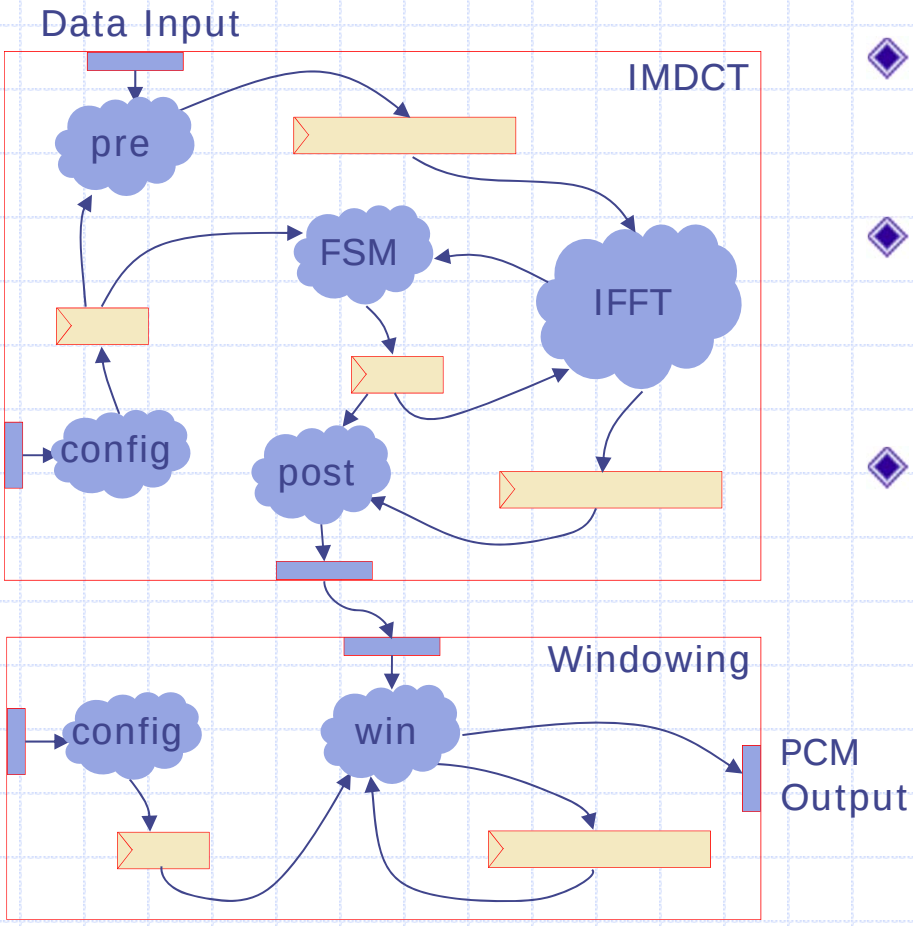
Example: Ogg Vorbis



- An audio compression format similar to MP3
- Front-end is naturally done in SW, back-end in HW
- IMDCT takes the most computation
- Is this the best partitioning choice?

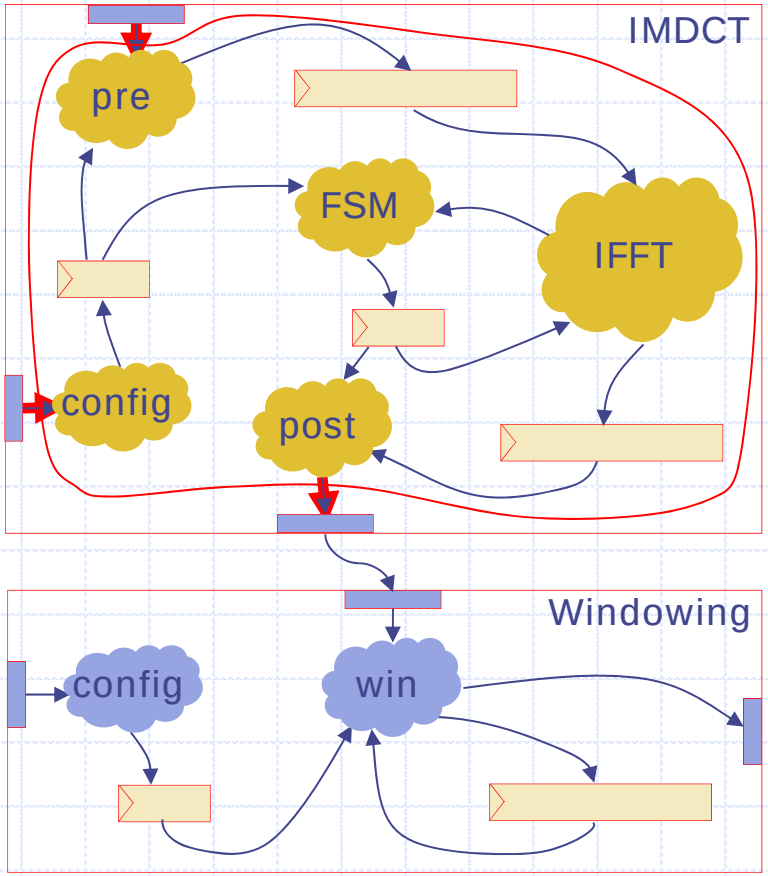


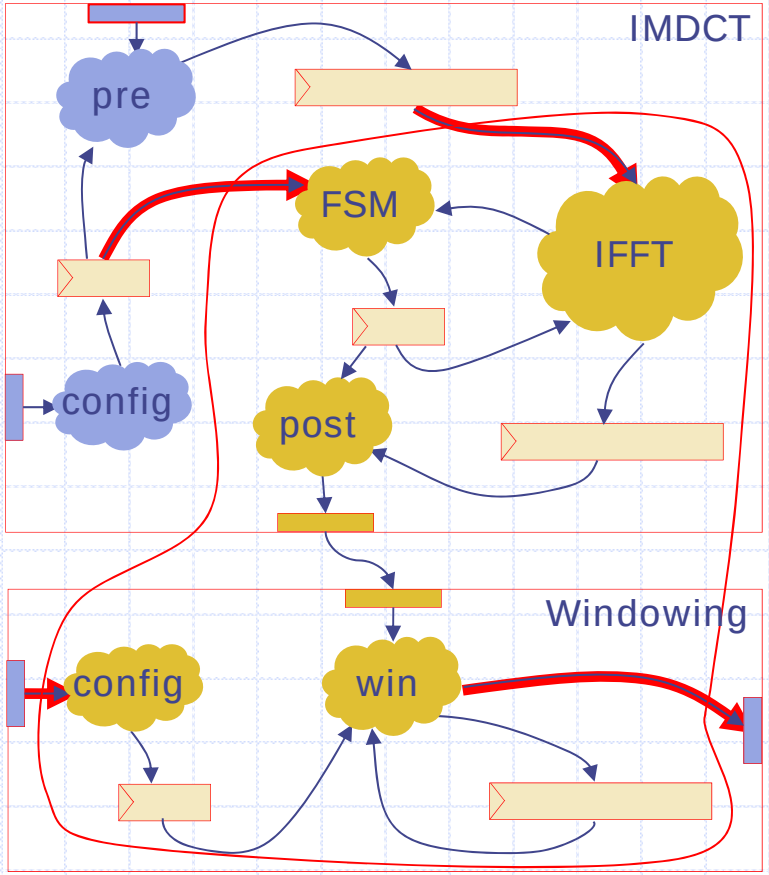
IMDCT + Windowing



- Actual Data-Flow for Vorbis back-end
- Edge traffic and computational intensity vary greatly
- Many options for partitioning these modules

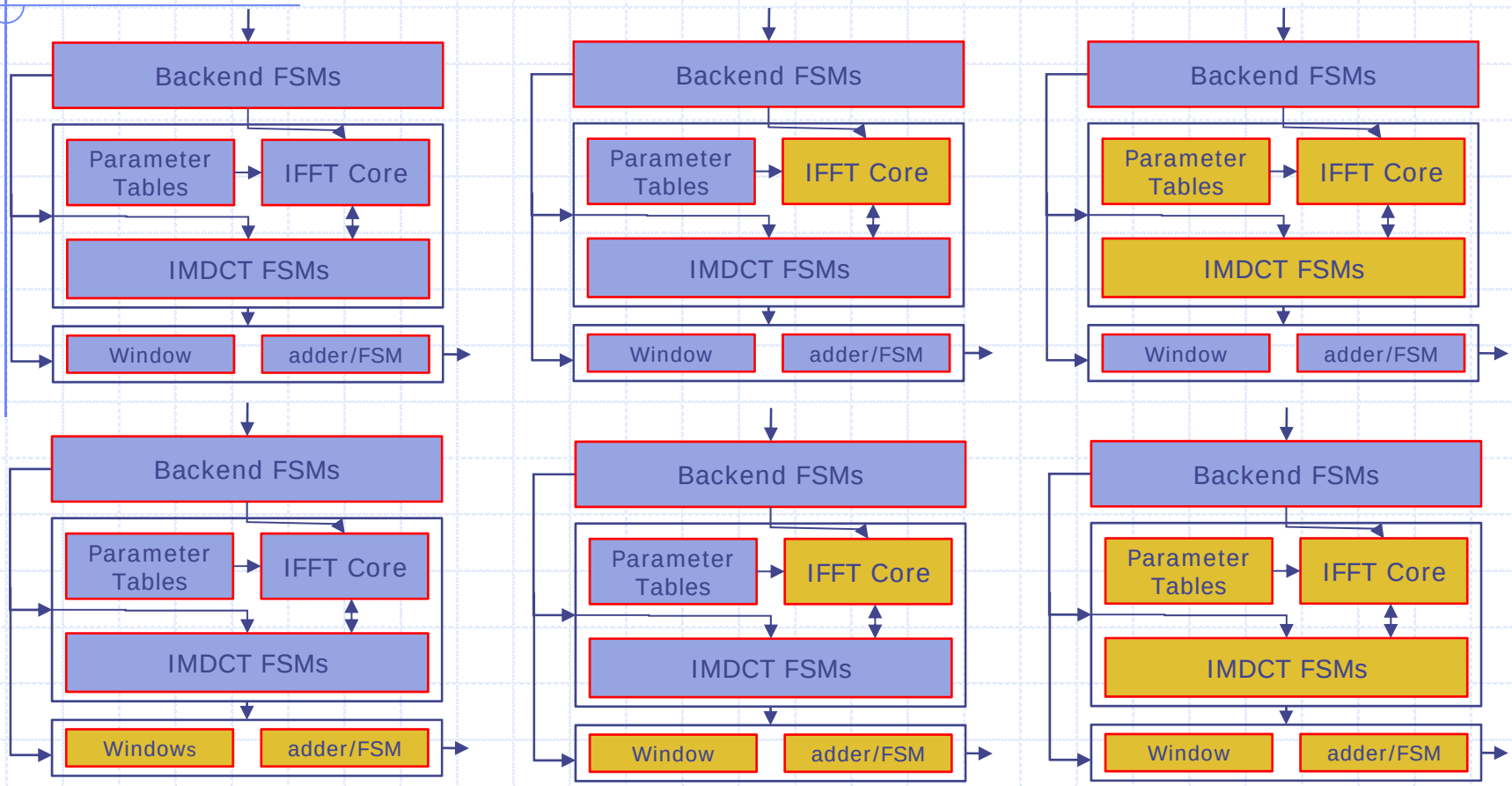
Partitioning Dictates Interface definition (Ifc)





If you fix the ifc first then very little choice is left for partitioning

many more choices...



Each partitioning results in vastly different amounts of data transfer across the HW-SW boundary which may over shadow computational acceleration

Why exploring partitioning is so difficult

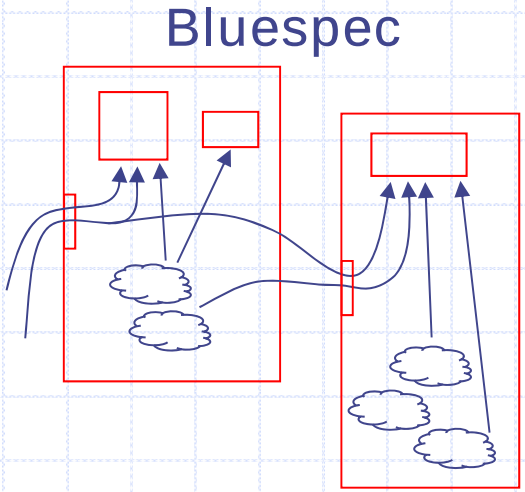
- Inflexible interface definitions
- Entirely different languages for expressing HW (e.g., Varilog) and SW (e.g., C, C++)
  - different programming/design cultures in the two worlds
- Complicated debugging environment

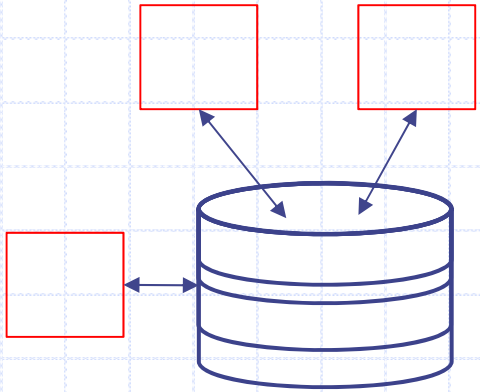
Solution: Express both HW and low-level SW in the same language,
e.g., Bluespec Codesign Language (BCL)

Outline

- Need for special purpose HW √
- Bluespec and compiling rules into HW and SW
- Partition Program into *Computational Domains*
- Automatically synthesize the *Interface* to connect the domains
- Evaluation

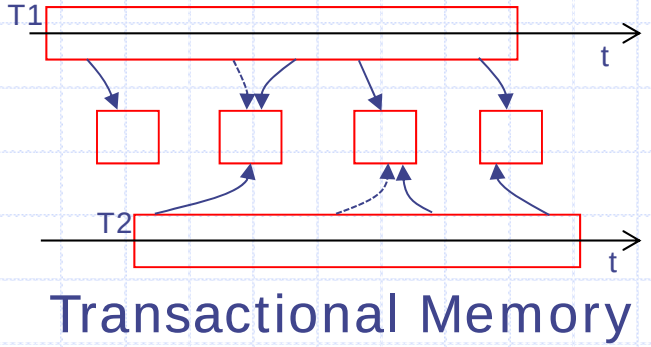
Bluespec





Bluespec is a transactional system;
All behavior is expressed in terms of *guarded atomic actions*, known as Rules

Rule :: Guard → Action



Transactions is proven model of parallel computation

Expressing computation using rules

```
int s = s0;                    C-code
for (int i = 0; i < 32; i = i+1)
    {s = f(s);} return s;
```

- Instantiate *state elements*

```
Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);
```

make a 6-bit register with initial value 32

- *Rules* define how state is to be transformed atomically

```
rule step if (i < 32);
  s <= f(s);
  i <= i+1;
endrule
```

the rule can execute only when its guard is true

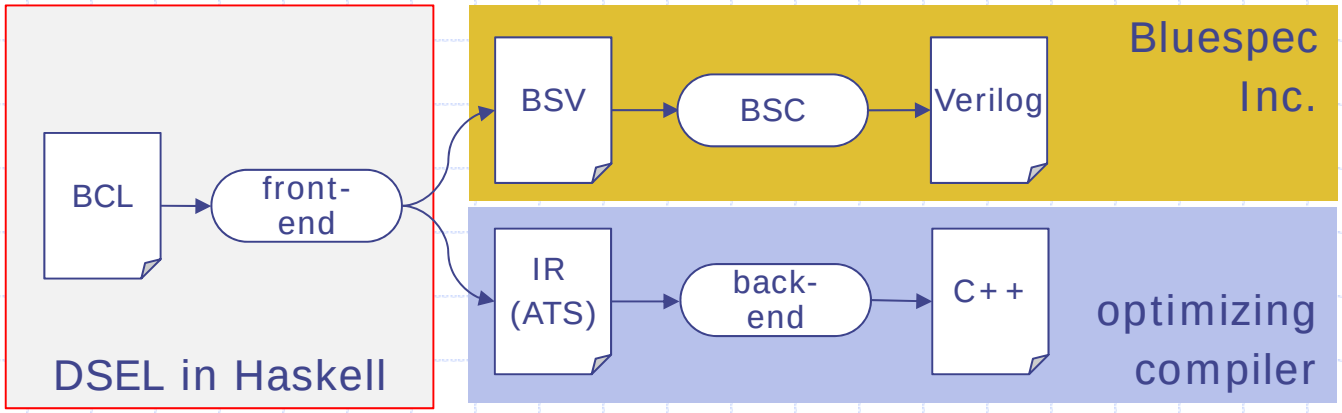
actions to be performed when the rule executes

- *Methods* define the interface

```
method start(s0) if (i==32);
   s <= s0;
endmethod
```

Compiling BCL

- BCL is an extension of Bluespec SystemVerilog (BSV) for expression efficient SW and partitioning



- Synthesizing the same rule system for both HW and SW
  - Parallel vs. Multithreaded sequential substrates

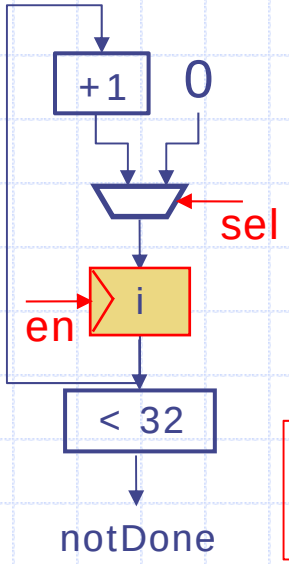
How do we generate efficient SW?

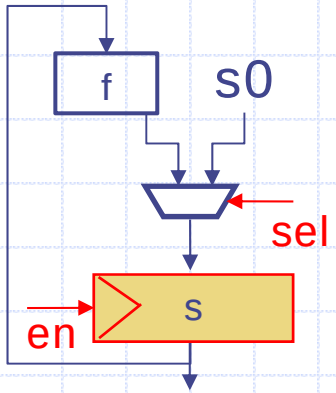
Myron King, Nirav Dave and Arvind, ASPLOS 2012

Rule Execution in HW

- When a rule executes:
  - all the registers are read at the beginning of a clock cycle
  - the guard and computations to evaluate the next value of the registers are performed
  - at the end of the clock cycle registers are updated iff the guard is true
- Muxes are need to initialize the registers

```
Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);
rule step if (i < 32);
     s <= f(s);
     i <= i+1;
endrule
```





sel = start
en = start | notDone

Challenges in Implementing Rules in SW

- Unlike hardware we need to make shadow state (copies) to deal with guard failures
- Efficient HW rule scheduling and SW rule scheduling are completely different
  - HW scheduling is well understood
- Optimizations for SW generation
  - Sequentialize parallel actions
  - Guard lifting for early failure detection
  - Partial shadowing of state

STM vs. Rules

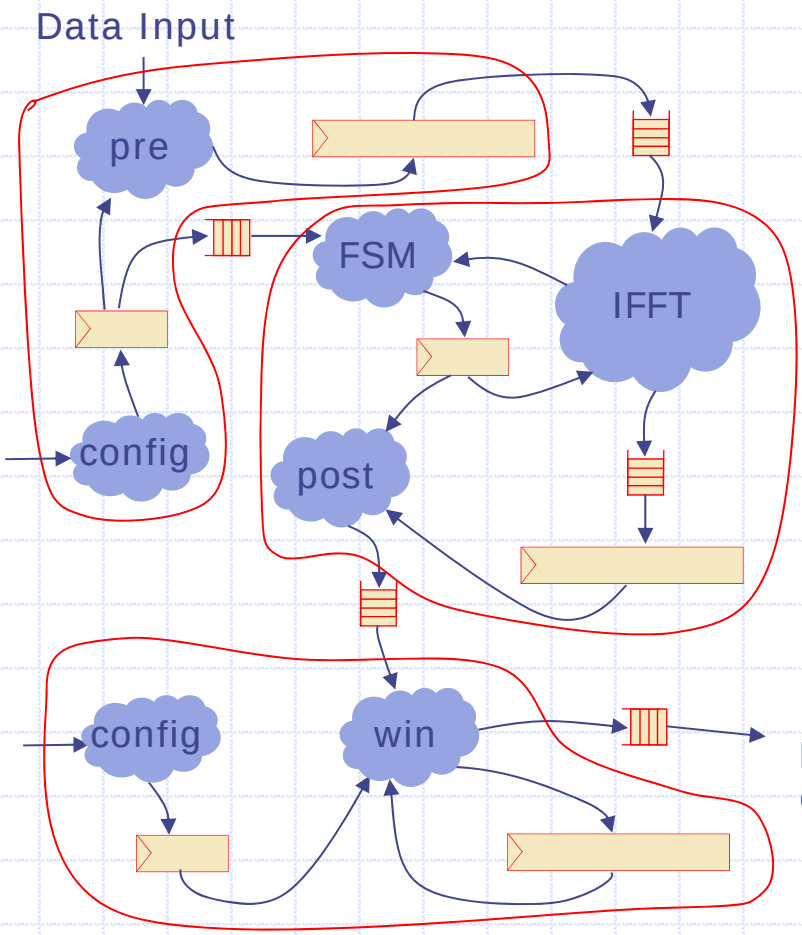
- Both modify state atomically and require linearizability of transactions
- We only schedule non-conflicting transactions simultaneously ⇒ no need to keep track of read sets and write sets
- Our Rules (transactions) fail *only* because of a *guard* failure

  ⇒ shadow state is required like in STM

Outline

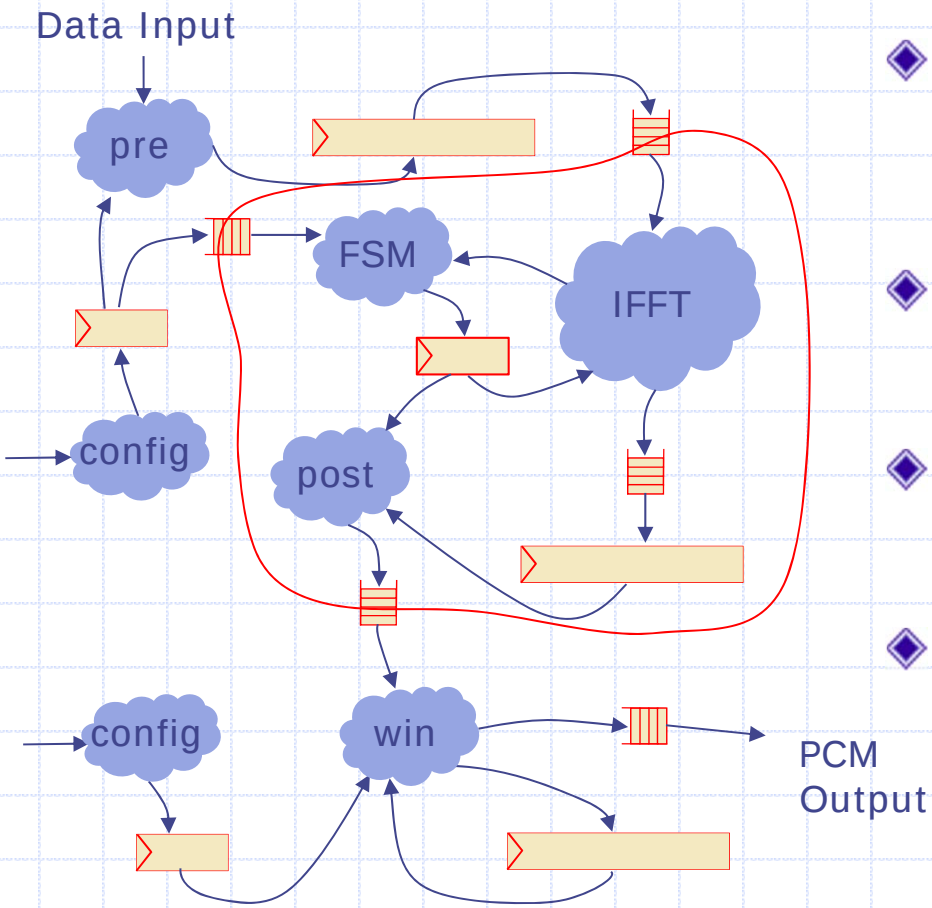
- Need for special purpose HW √
- Bluespec and compiling rules into HW and SW √
- Partition Program into *Computational Domains*
- Automatically synthesize the *Interface* to connect the domains
- Evaluation

Computational Domains: Where to partition



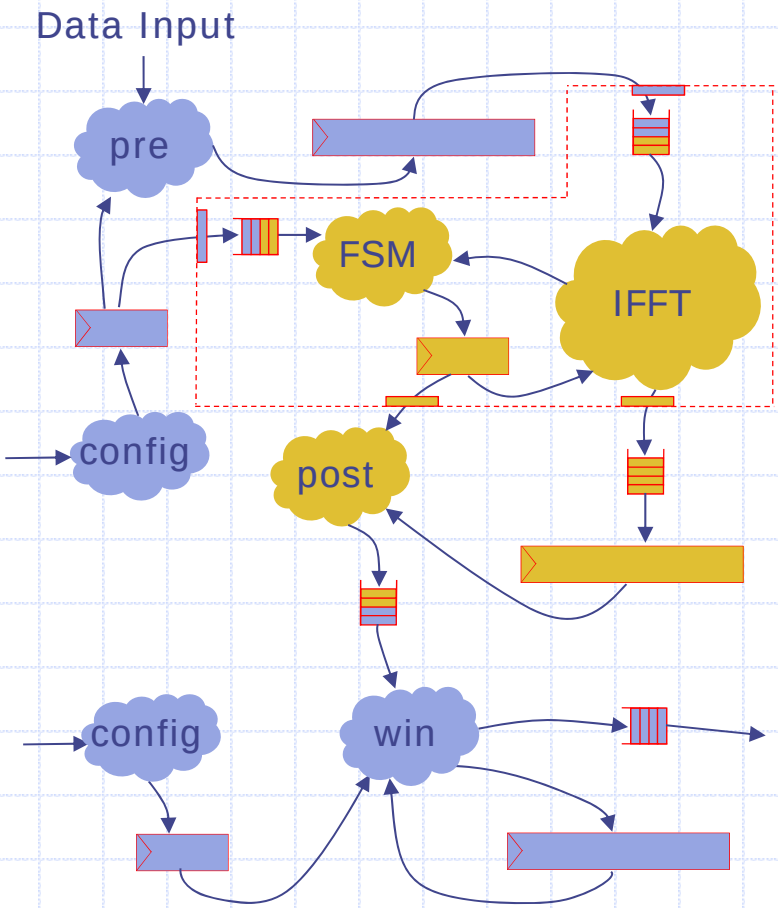
- It is easy to implement a FIFO abstraction between HW and SW
- Shared registers/variables introduce coherence issues (excessive synchronization)
- Group functionality into *computational domains* connected by FIFOS
- Design styles which enable modular refinement partition naturally

Enforcing Safe Partitions



- Every rule in a computational domain gets a color
- Rules and registers are mono-chromatic
- Domain crossing FIFOs have two colors
- Enforced by the type system in BCL

Partitioning and Modularity



- Modularity and Partitioning are orthogonal. Module structure may reflect partitioning requirements
- IFFT interface declaration:

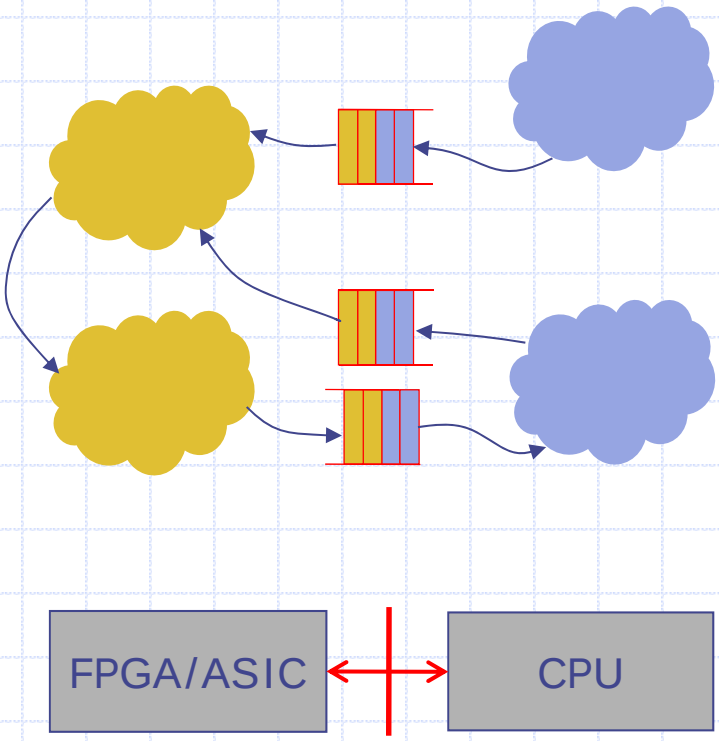
```
data (IFFT m n t) = IFFT{
 input  ::
   Method m (t -> Action)
 config ::
   Method m (State -> Action)
 output ::
   Method n (ActionValue t)
 status ::
   Method n (State)
}
```

m and n are domain type variables; to be specified in the implementation

Outline

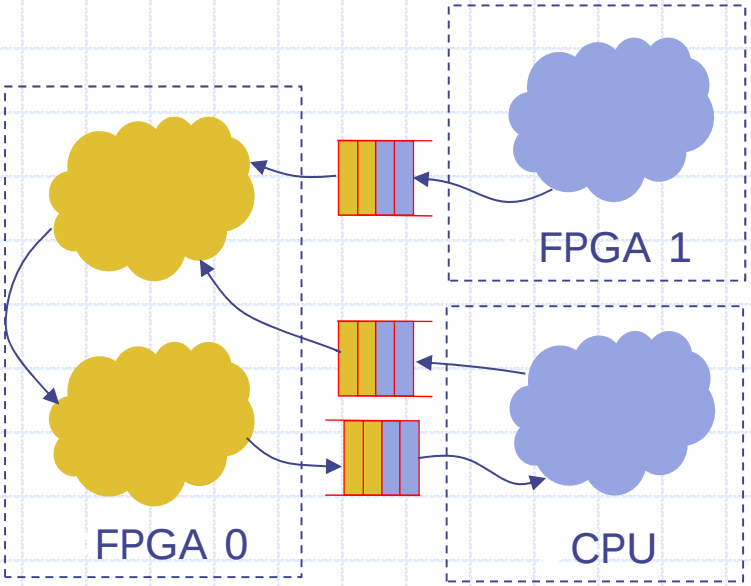
- Need for special purpose HW √
- Bluespec and compiling rules into HW and SW √
- Partition Program into *Computational Domains* √
- Automatically synthesize the *Interface* to connect the domains
- Evaluation

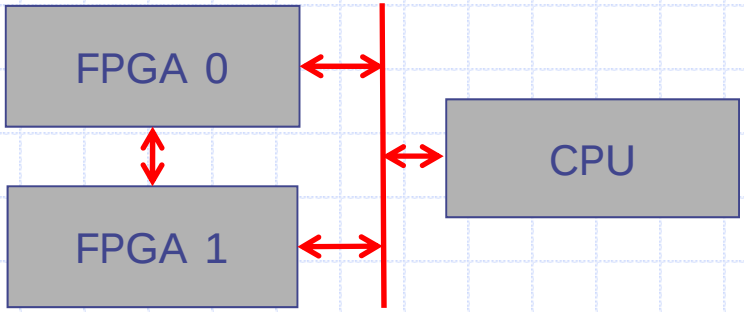
Mapping Domains



- Domains form *Latency Tolerant BDN*
- HW and SW substrates form a *Physical Network* (PN) of partitions
- LT-BDN must be mapped to the PN
- FIFO traffic (rate) is not statically known ➔ automated dynamic scheduling

Multiplexing shared physical channel





Bus (PCI Express)

- One *Virtual Channel* per FIFO with flow-control and sufficient buffering
  - Dally and Seitz, 1987
- HW/SW connection logic is automatically generated
- All data-types are automatically marshaled and translated

Outline

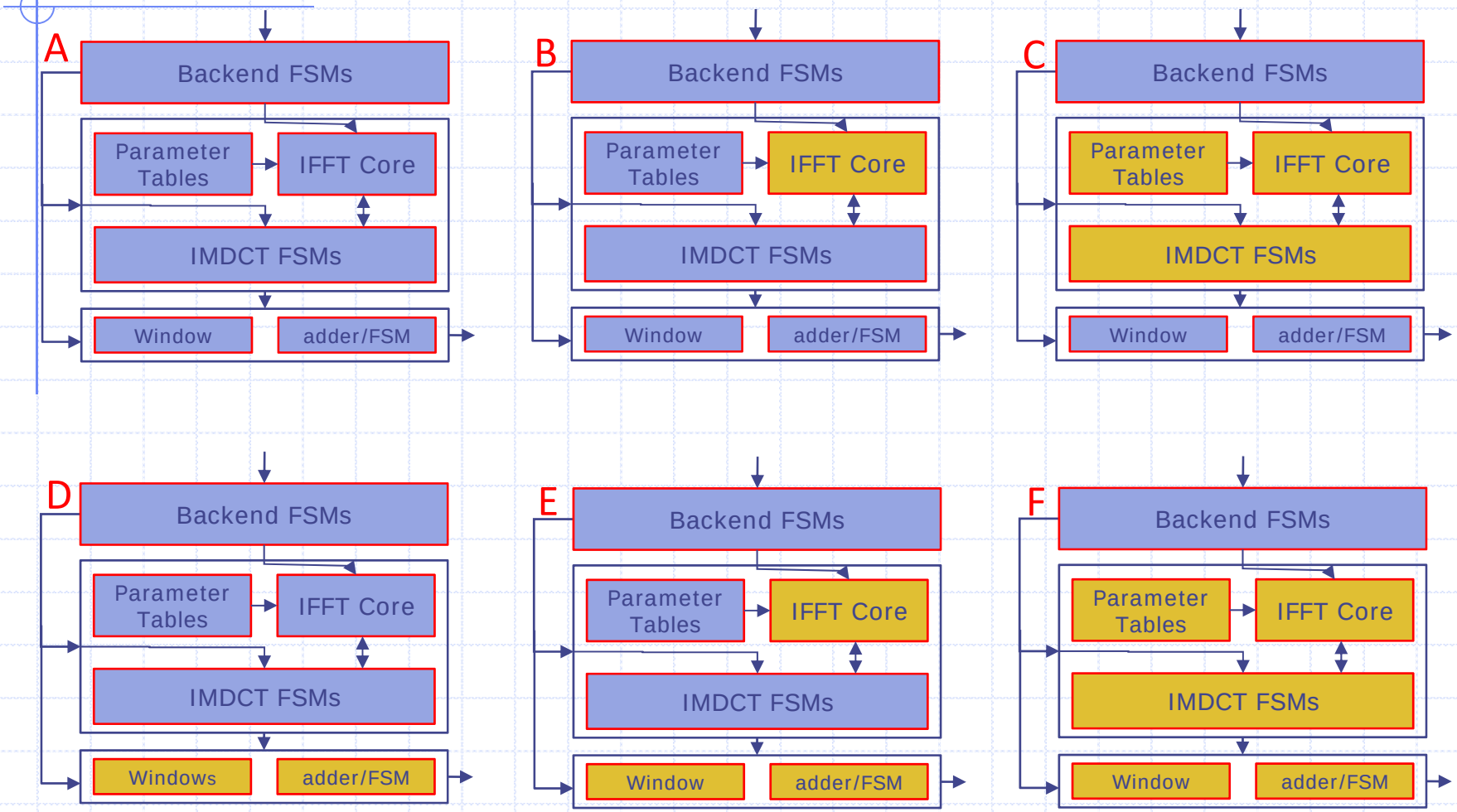
- Need for special purpose HW √
- Bluespec and compiling rules into HW and SW √
- Partition Program into *Computational Domains* √
- Automatically synthesize the *Interface* to connect the domains √
- Evaluation

Evaluation

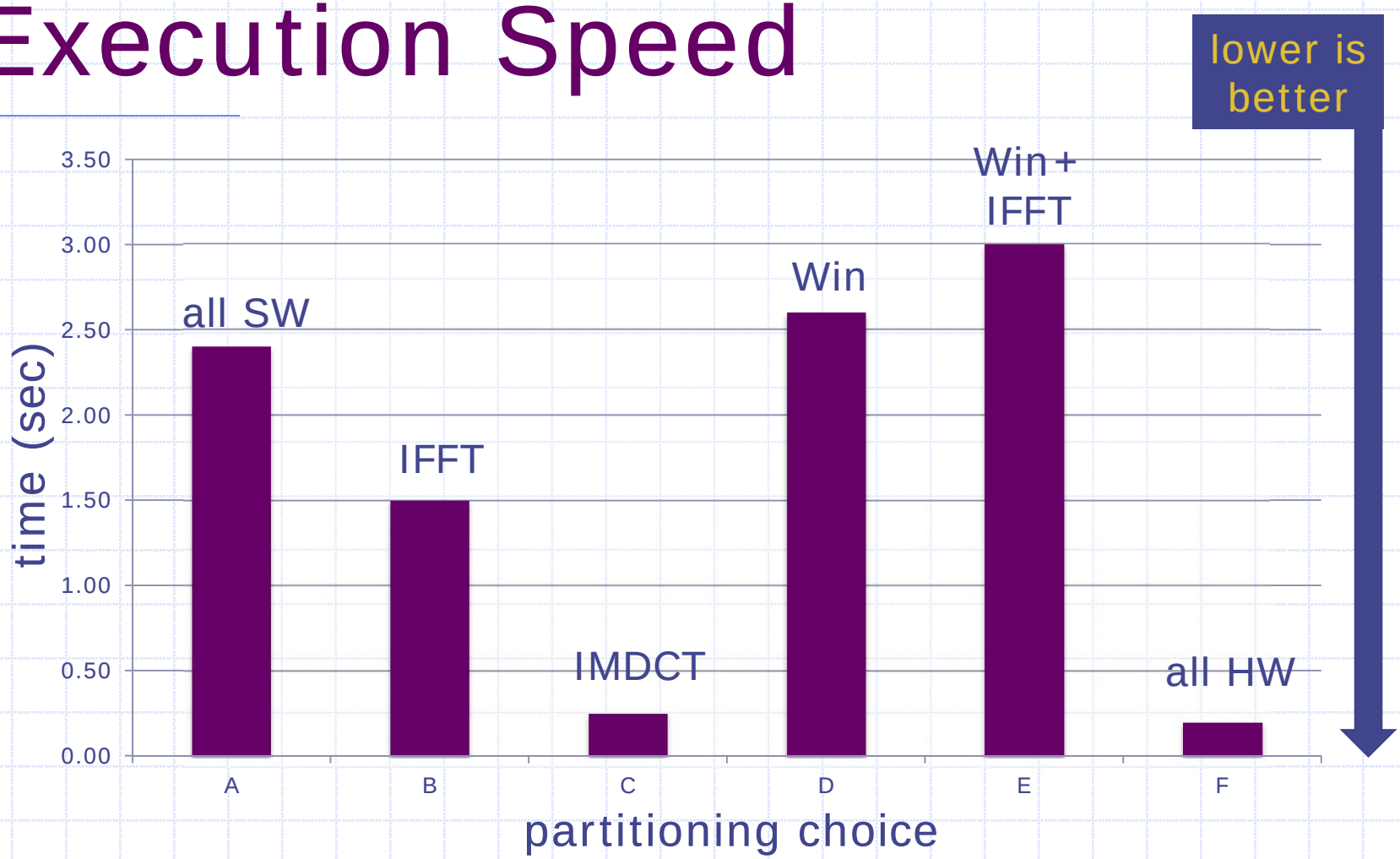
- Benchmarks: many different partitions
  - Ogg Vorbis
  - Ray Tracing
  - EEMBC benchmark suite & many more
- Xilinx ML507: XC5VFX70T chip

  - PowerPC 440 (400 MHz)
  - FPGA Fabric (100 MHz)
  - 256MB DDR2

Partitioning Vorbis back-end



Execution Speed



the results agree with our intuition

Related Work

- Implementation-agnostic parallel models
  - *Hoe and Arvind (2000)*
  - Chandy and Misra (Unity, 1988)
  - Dijkstra (Guarded Commands, 1975)
- Generation of SW from HW Descriptions
  - Chiou et al. (Chinook, 1995)
  - Any optimized RTL simulator
- Generation of HW from Seq. SW Description
  - CatapaultC, Pico Platform, AutoPilot (commercial)
  - Huang et al. (Liquid Metal 2008)
- Simulating heterogeneous systems
  - Buck et al. (Ptolemy 1994)
  - Balarin et al. (Metropolis 2003)
- Algorithms for HW/SW Partitioning
  - A lot of work in this area…

Takeaway

- Power concerns require special purpose hardware in all SoCs
- HW/SW Codesign is challenging
  - *tedious and error prone*
  - *rewriting inhibits design exploration*
- Use a unified language for both HW and SW
  - partitioning can be specified in the source code as a simple coloring scheme
  - both HW and SW can be generated from the same source code
  - interfaces and communication infrastructure can be synthesized automatically

Thank you