MP SoC Summer School
8 –12 June 2002

Communication as the backbone for a well balanced system design

Eric.Verhulst@eonic.com

Eonic Solutions GmbH, Germany

www.eonic.com

The von Neumann ALU versus an embedded processor

- The sequential programming paradigm is based on the von Neumann architecture
- But this was only meant for one ALU
- A real processor in an embedded system :
  - Inputs data
  - Processes the data : only this covered by von Neumann
  - Output the result
- On other words : at least two communications, often one computation
- => Communication/Computation ratio must be > 1 (in optimal case)
- Standard programming languages (C, Java, …) only cover the computation and sometimes limited runtime multitasking
- Conclusion :
  - We have an unbalance, and have been living with it for decades
- Reason ? : history
  - Computer scientists use workstations
  - Only embedded systems must process data in real-time
  - Embedded systems were first developed by hardware engineers

Multi-tasking

- Origin :
  - A software solution to a hardware limitation
  - von Neumann processors are sequential, the real-world is “parallel” by nature and software is just modeling
  - Developed out of industrial needs
- How to ?
  - A function is a [callable] sequential stream of instructions
  - Uses resources [mainly registers] => defines “context”
  - Non-sequential processing =
    - switching between ownership of processor(s)
    - reducing overhead by using idle time or to avoid active wait :
      - each function has its own workspace
      - a task = function with proper context and workspace
    - Scheduling to achieve real-time behavior for each task

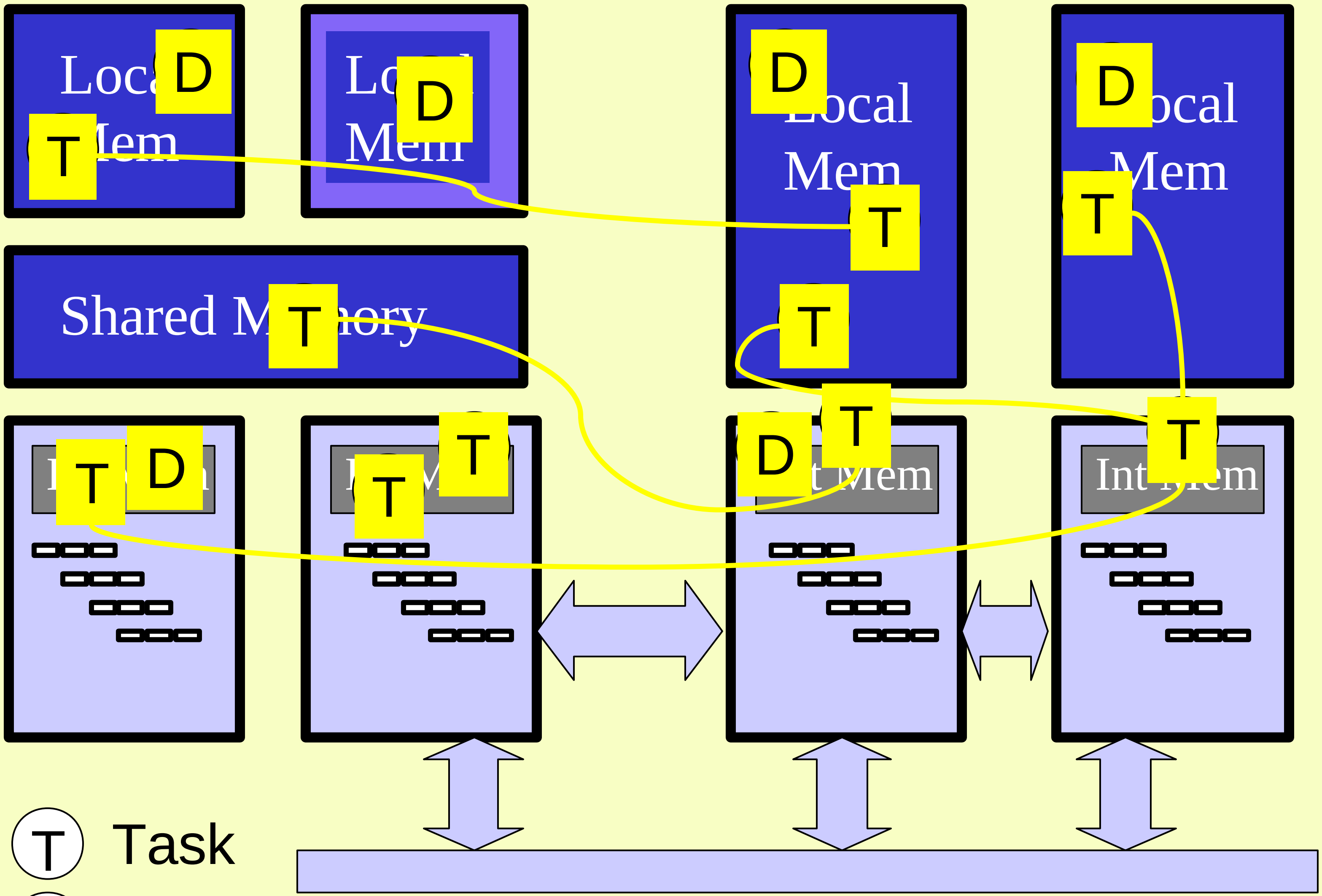
Scheduling algorithms

- Three dominant real-time/scheduling paradigms :
  - control flow :
    - event driven - asynchronous : latency is the issue
    - traverse the state machine
    - uncovered states generate complexity
  - data-flow :
    - data-driven : throughput is the issue
    - multi-rate processing generates complexity
  - time-triggered :
    - play safe : allocate timeslots beforehand
    - reliable if system is predictable and stationary
  - REAL SYSTEMS :
    - combination of above
    - distinction is mainly implementation and style issue, not conceptual
    - SCHEDULING IS AN ORTHOGONAL ISSUE TO MULTI-TASKING

Why Multi-Processing ?

- Laws of diminishing return :
  - Power consumption increases more than linearly with speed
  - Highest speed achieved by micro-parallel tricks :
    - Pipelining, VLIW, out of order execution, branch prediction, …
    - Efficiency depends on application code
  - Requires higher frequencies and many more gates
  - Creates new bottlenecks :
    - I/O and communication become bottlenecks
    - Memory access speed slower than ALU processing speed
- Result :
  - 2 processors @1F Hz can be better than one @2F Hz if communication support (HW and SW) is adequate
- The catch :
    - Not supported by von Neumann model
    - Scheduling, task partitioning and communication are inter-dependent
    - BUT SCHEDULING IS NOT ORTHOGONAL TO PROCESSOR MAPPING AND INTERPROCESSOR COMMUNICATION

Generic MP system



T Task

D data

A task is more

- Tasks need to interact
  - synchronize
  - pass data = communicate
  - share resources
- A task = a virtual single processor or unit of abstraction
- A (SW) multi-tasking system can emulate a (HW) real system
- Multi-tasking needs communication services
- Theoretical model :
  - CSP : Communicating Sequential Processes (and its variations)
  - C.A.R. Hoare
  - CSP := sequential processes + channels
  - Channels := synchronised (blocked) communication, no protocol
  - Formal, but doesn't match complexity of real world
- Generic model : module based, multi-tasking based, process oriented ,…
  - Generic model matches reality of MP-SoC
  - Very powerful to break the von-Neumann constrictor

There is only programs

- Simplest form of computation is assignment :

a:= b

- Semi-Formal :

```
BEFORE        : a = UNDEF;        b = VALUE(b)
AFTER         : a = VALUE(b);     b = VALUE(b)
```

- Implementation in typical von Neumann machine :

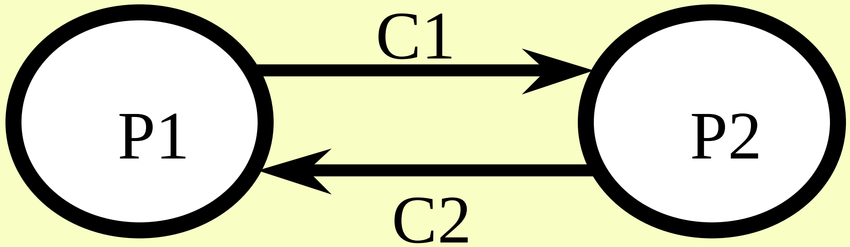
```
Load b, register X
Store X, a
```

CSP explained in occam

```
PROC P1, P2 :
 CHAN OF INT32 c1,c2 :

 PAR
   P1(c1, c2)
   P2(c1, c2)
```

```
/* c1 ? a : read from channel c1 into variable a */
/* c2 ! b : write variable b into channel c2 */
/* order of execution not defined by clock but by */
/* channel communication : execute when data is ready */
```



Needed :

- context
- communication

A small parallel program

No assumption in PAR case about order of execution => self-synchronising

P1

```
INT32 a :

SEQ
  a:= ANY
  c1 ! a
```

c1

P2

```
INT32 b :

SEQ
  b:= ANY
  c1 ? b
```

Equivalent :

```
SEQ
  INT32 a,b :
  a:= ANY
  b:= ANY
  b:= a
```

The PAR version at von Neumann machine level

- PROC_1

```
Load b, register X
Store X, output register
(hidden : start channel transfer)
(hidden : transfer control to PROC_2)
                          /*Single Processor*/
```

- PROC_2

```
(hidden : detect channel transfer)
(hidden : transfer control to Proc_2)
Load input register, X
Store X, b
```

- In between :
  - Data moves from output register to input register
  - Sequential case is an optimization of the parallel case

The same program for hardware with Handel-C

```
Void main(void)
par /* WILL GENERATE PARALLEL HW (1 clock cycle) */
  chan chan_between;
  int a, b;
  { chan_between ! a
    chan_between ? b
  }
```

But :

```
Seq /* WILL GENERATE SEQUENTIAL HW (2 clock cycles) */
  chan chan_between;
  int a, b;
  chan_between ! a
  chan_between ? b
  }
```

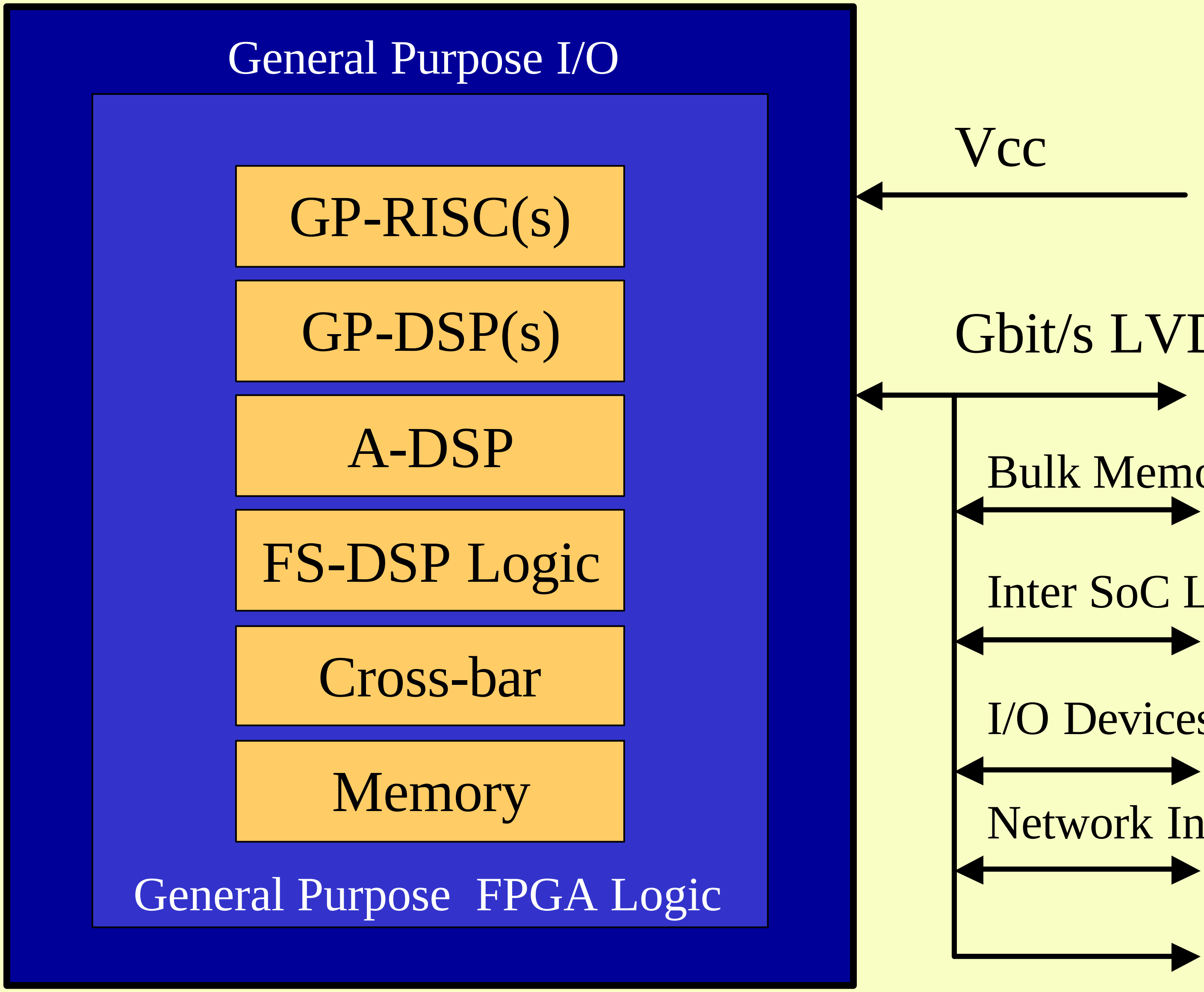
Consequences

- Data is protected inside scope of process
- Interaction is through explicit communication
- For HW design :
  - In order to safeguard abstract equivalence :
    - Communication backbone needed
    - Automatic routing needed (but deadlock free)
    - Process scheduler if on same processor
  - In order to safeguard real-time behavior
    - Prioritisation of communication for dynamic applications
    - Allocate time-slots beforehand for stationary applications
  - In order to handle multi-byte communication :
    - Buffering at communication layer
    - Packetisation
    - DMA in background
  - Result :
    - prioritized packet switching : header, priority, payload
    - Communication not fundamentally different from data I/O

Future chips becoming SoC

- High NRE, high frequency signals
- Conclusion :
  - multi-core, course grain asynchronous SoC design
  - cores as proven components -> well defined interfaces
  - keep critical circuits inside
  - simplify I/O, reduce external wires :
    - high speed serial links, no buses
  - NRE dictates high volume -> more reprogramability
  - system is now a component
  - below minimum thresholds of power and cost, it becomes cheap to "burn" gates
  - software becomes the differentiating factor

The (next generation) SoC



Early examples

- Board level : adoption of "switch fabrics" for telecom
  - SpaceWire (IEEE1355) : in use at CERN, ESA, …
  - PICMG 2.16 … 2.20
  - PICM 3.xx (AdvancedTCA)
- Motorola e500
  - Based on RapidIO
  - On-chip switch
  - Complex due to throwing together memory addressing and link comm
- Xilinx VirtexII-Pro (available)
  - Aurora links (3.4 Gbit/sec, user programmable link layers, protocols)
  - Up to 4 PPC inside + softcore CPU
- Altera Stratix
  - Links, memory
  - ARM and softcore CPU

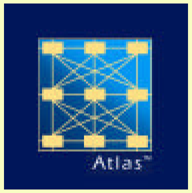
Beyond multi-tasking in C

- Multi-tasking = Process Oriented Programming
- A Task =
  - Unit of execution
  - Encapsulated functional behavior
  - Modular programming
- High Level [Programming] Language :
  - common specification :
    - for SW
      - compile to asm
    - for HW
      - compile to VHDL or Verilog
  - E.g. program PPC with ANSI C (and RTOS), FPGA with Handel-C
  - C level design is enabler for SoC “co-design”
    - More abstraction gives higher productivity
    - But interfaces be better standardized for better re-use
    - Interfaces can be “compiled” for higher volume applications

Next : Virtual Single Processor (VSP) model

- Transparent parallel programming
  - Cross development on any platform + portability
  - Scalability, even on heterogeneous targets
- Distributed semantics
  - Program logic neutral to topology and object mapping
  - Clean API provides for less programming errors
  - Prioritized packet switching communication layer
- Based on “CSP” (C.A.R. Hoare): Communicating Sequential Processes: VSP is pragmatic superset
- Implemented first in Virtuoso VSP RTOS (now VSPWorks of Wind River)

Multitasking and message passing
Process oriented programming
Interfacing using communication protocols
Application doesn’t need to know physical layer

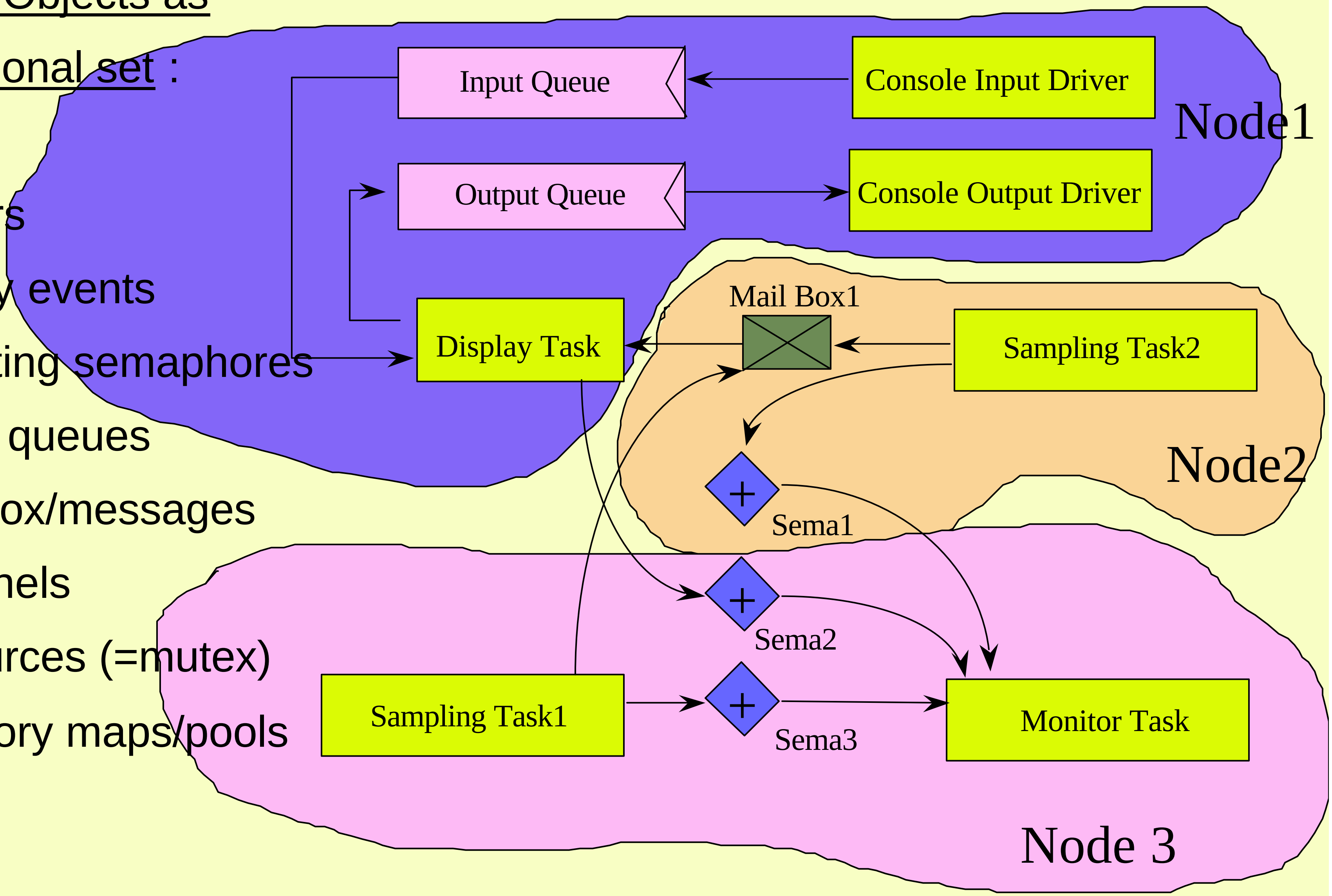


Virtuoso's Virtual Single Processor : a pragmatic CSP : distributed semantics

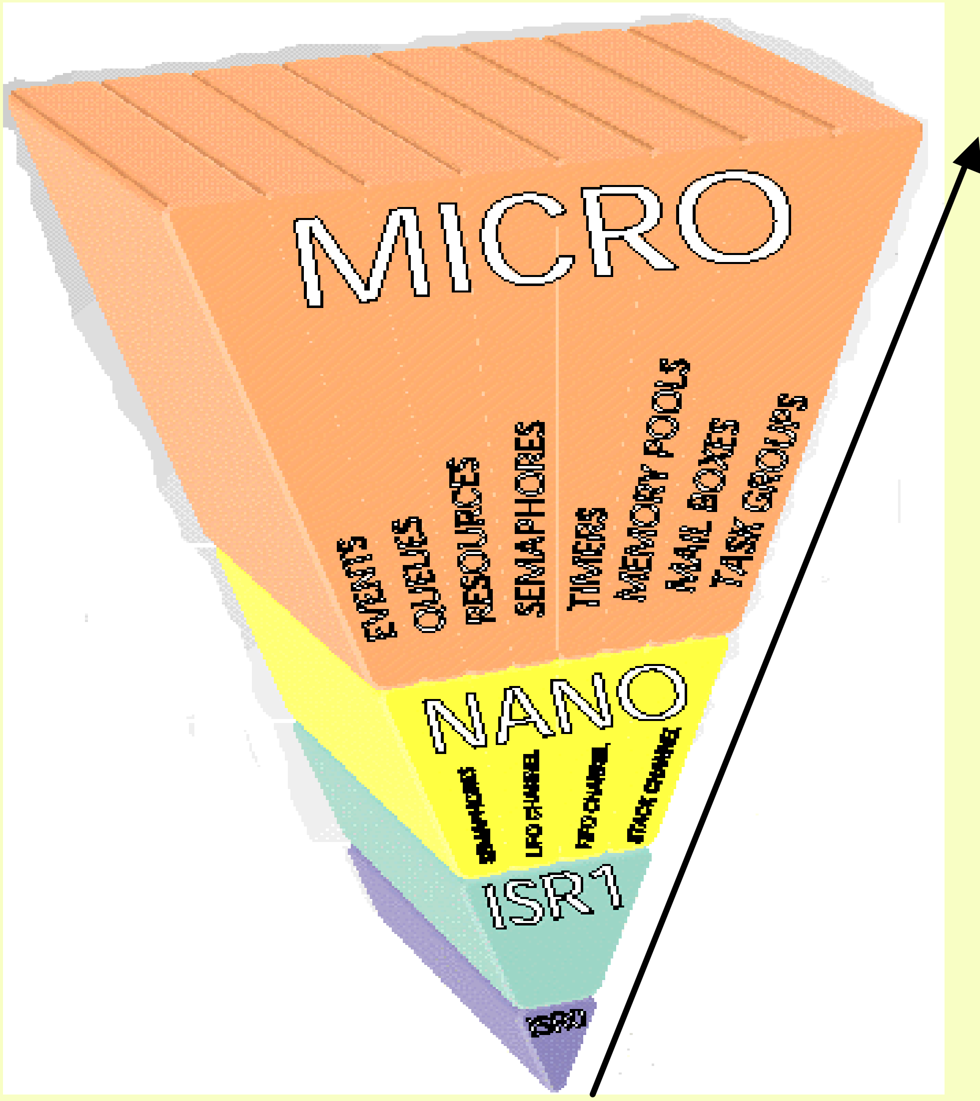
RTOS Objects as

Orthogonal set :

- tasks
- drivers
- binary events
- counting semaphores
- FIFO queues
- mailbox/messages
- channels
- resources (=mutex)
- memory maps/pools



Hierarchy and HW and time resources



Abstract behavior
Application level
SW flexibility
High Level Language
Register context
Memory use
System level
Latency
Data packet sizes
Hardware determinism

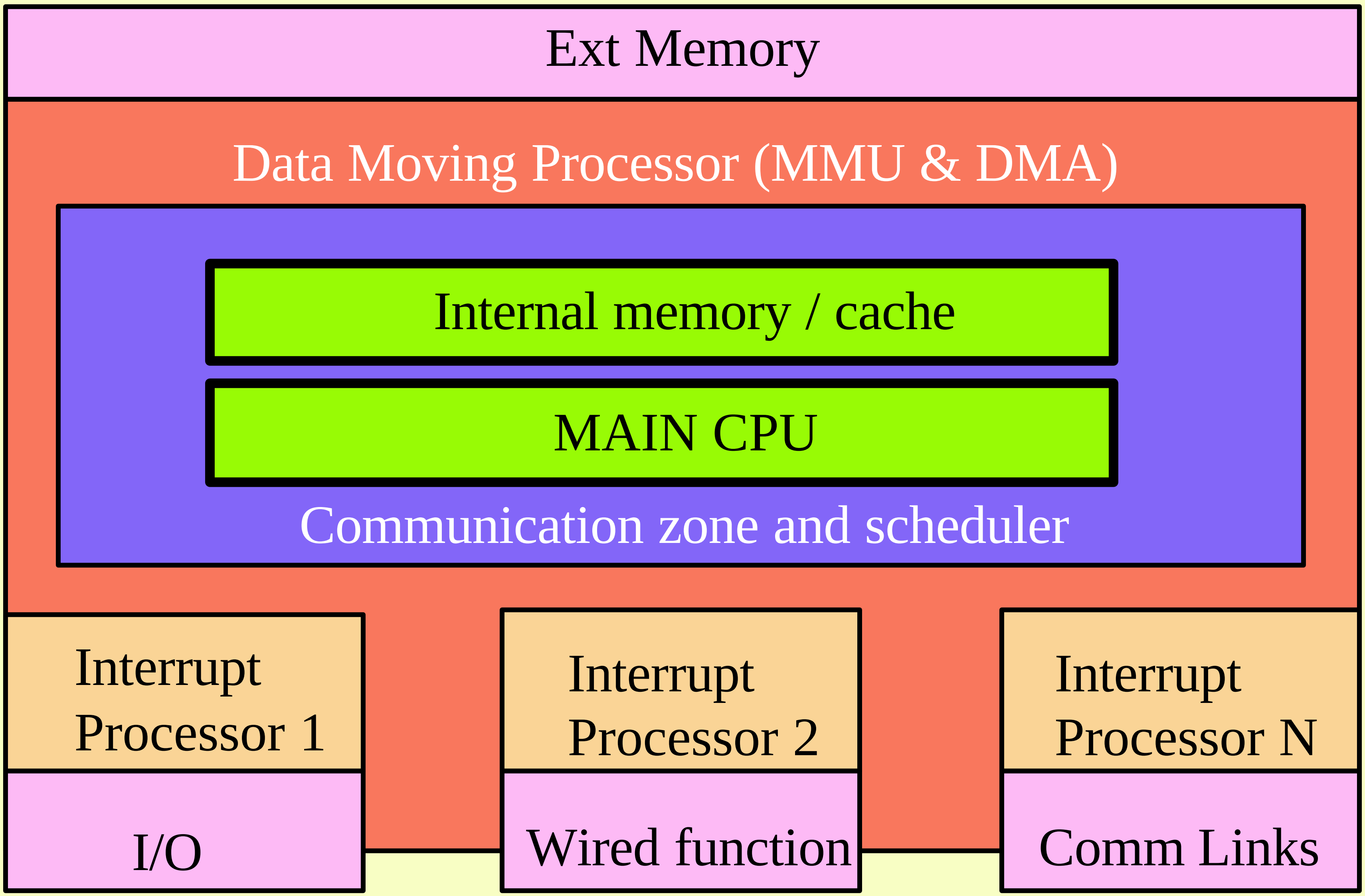
Mapping the RTOS architecture into HW

- On today's processors :
  - Assembler required (a lot of it !)
    - No or little support for context switching (+ obstacles)
    - No or elementary support for communication
- The functional layers of an application
  - I/O :
    - Interrupt processing ISR0 (2-4 regs)
    - Buffering data ISR1 (4-6 regs)
    - Drivers (atomic datamovers) Nanokernel process (8 regs)
    - NOTE : above can be pushed into co-processing hardware !
  - Processing :
    - Data driven : DSP Task & coprocessors (all regs)
    - Control driven : decision logic Task (global data)

The von Neumann state machine and its solution

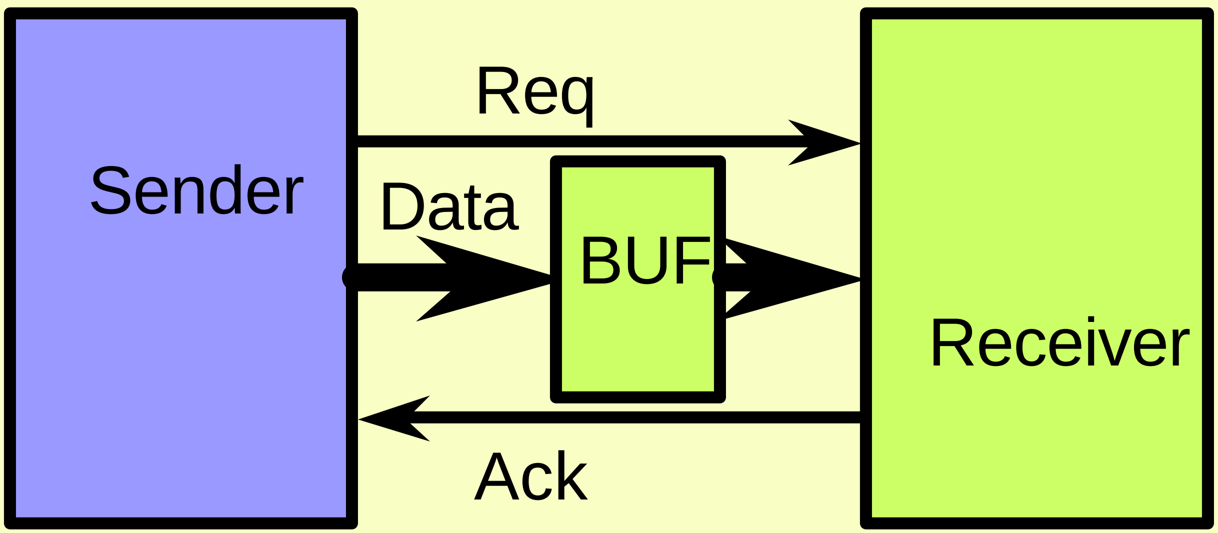
- Most processors are designed for throughput maximalisation
- Single CPU handles processing <u>and</u> I/O
- Large register context < > I/O & swapping
- I/O “engines” (if any) are special purpose
- Increasing bandwidth gap CPU-memory
- Result : large, complex state machine
- Solution :
  - parallel CSP architecture at the CPU level
  - Means : isolate the processing from the I/O
  - use “asynchronous” design techniques

A CSP based processor that is VSP friendly



CSP at the HW level

- Request/Ack protocol assures correct data transfer between async units, even at the register level
- Is like the mailbox mechanism



RTOS objects : mapping onto HW

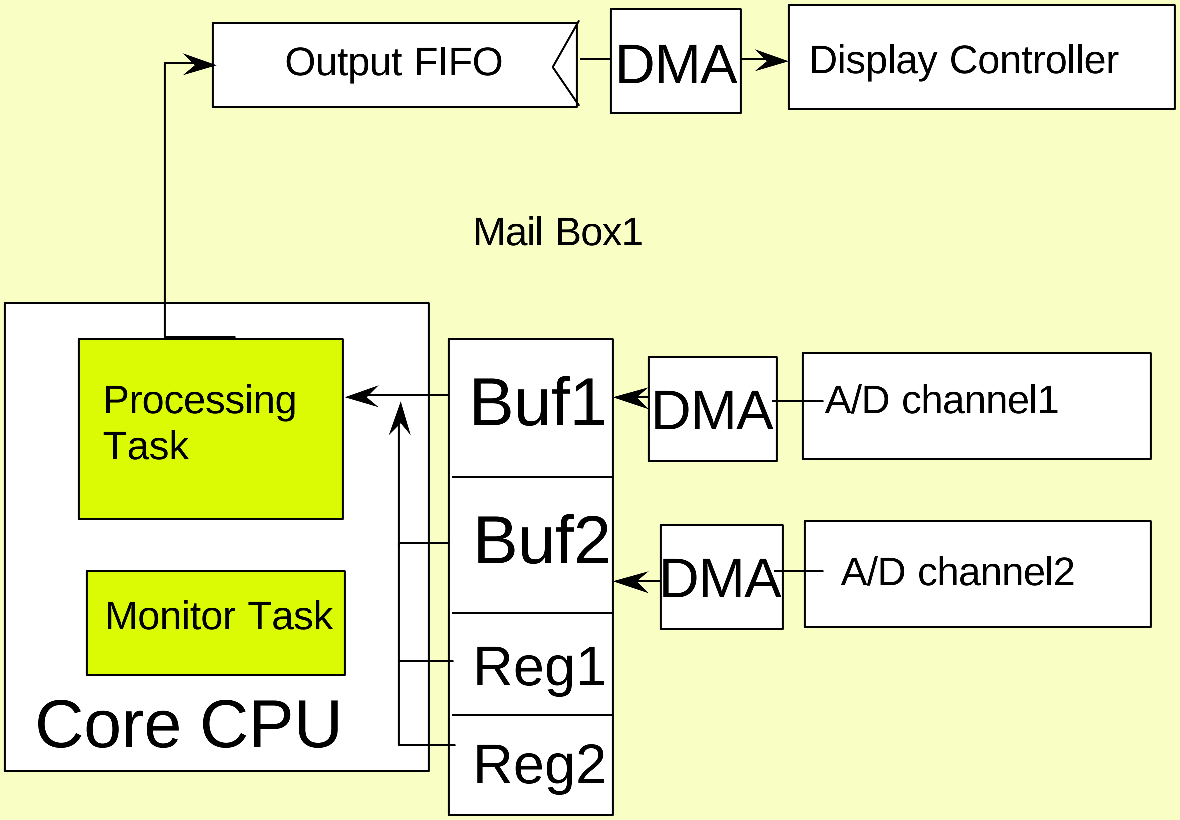
Software	Hardware
Task - Process	Logic State Machine
KS_FifoPutW	FIFO memory
KS_MsgPutW	shared memory + dma
KS_SemaSignal	status register + counter

RTOS objects can be used for SW+HW system specification, simulation and implementation

A SW-HW implementation (see slide 19)

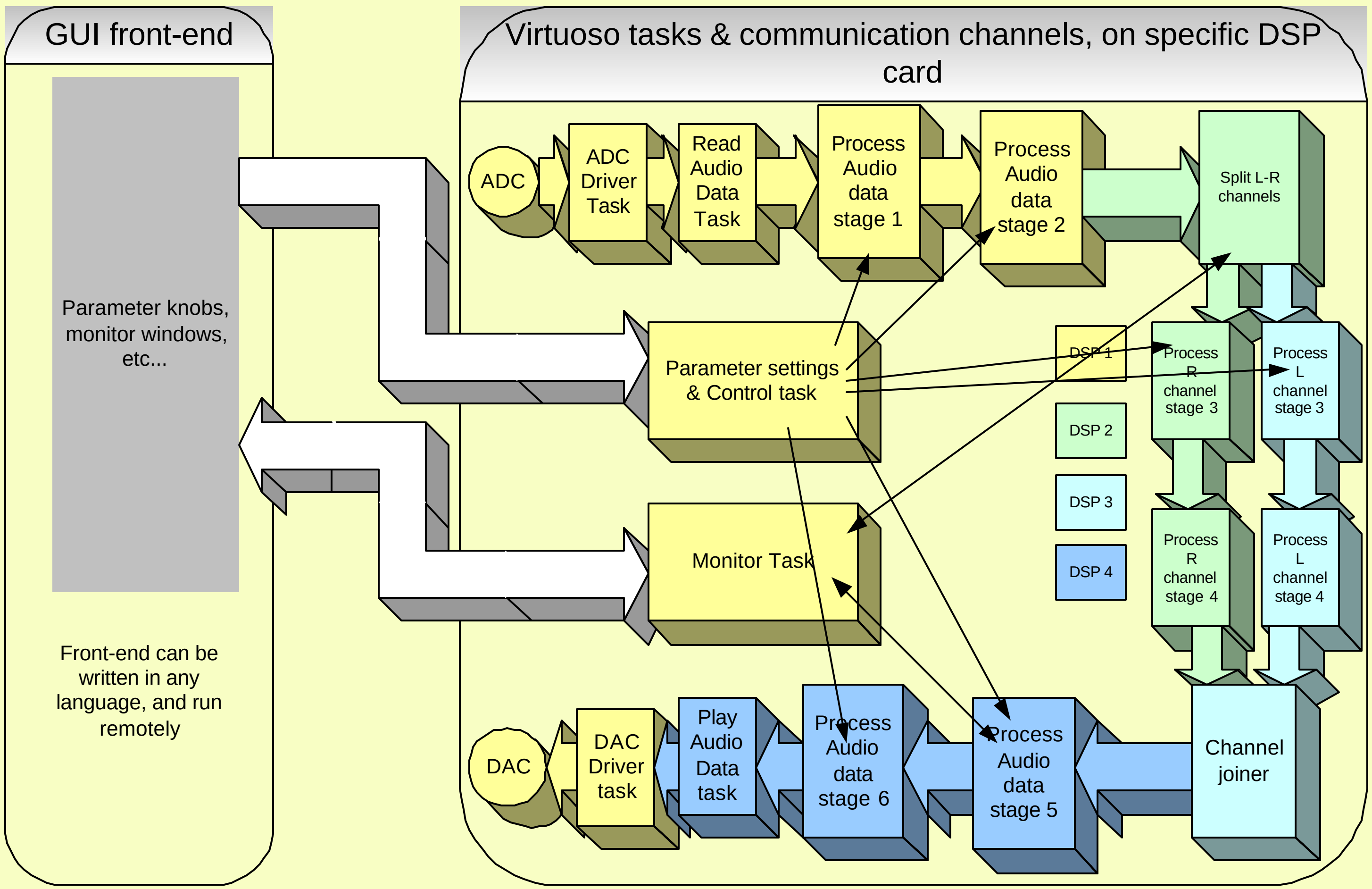
Steps :

1. Algorithm using MATLAB/ SDT, Pegasus, ...
2. Simulate logic model with RTOS simulator on host OS like NT
3. Run RTOS program on target CPU
4. Map parts onto SW (C to ASM - binary) map parts onto HW (C to VHDL or RTL)



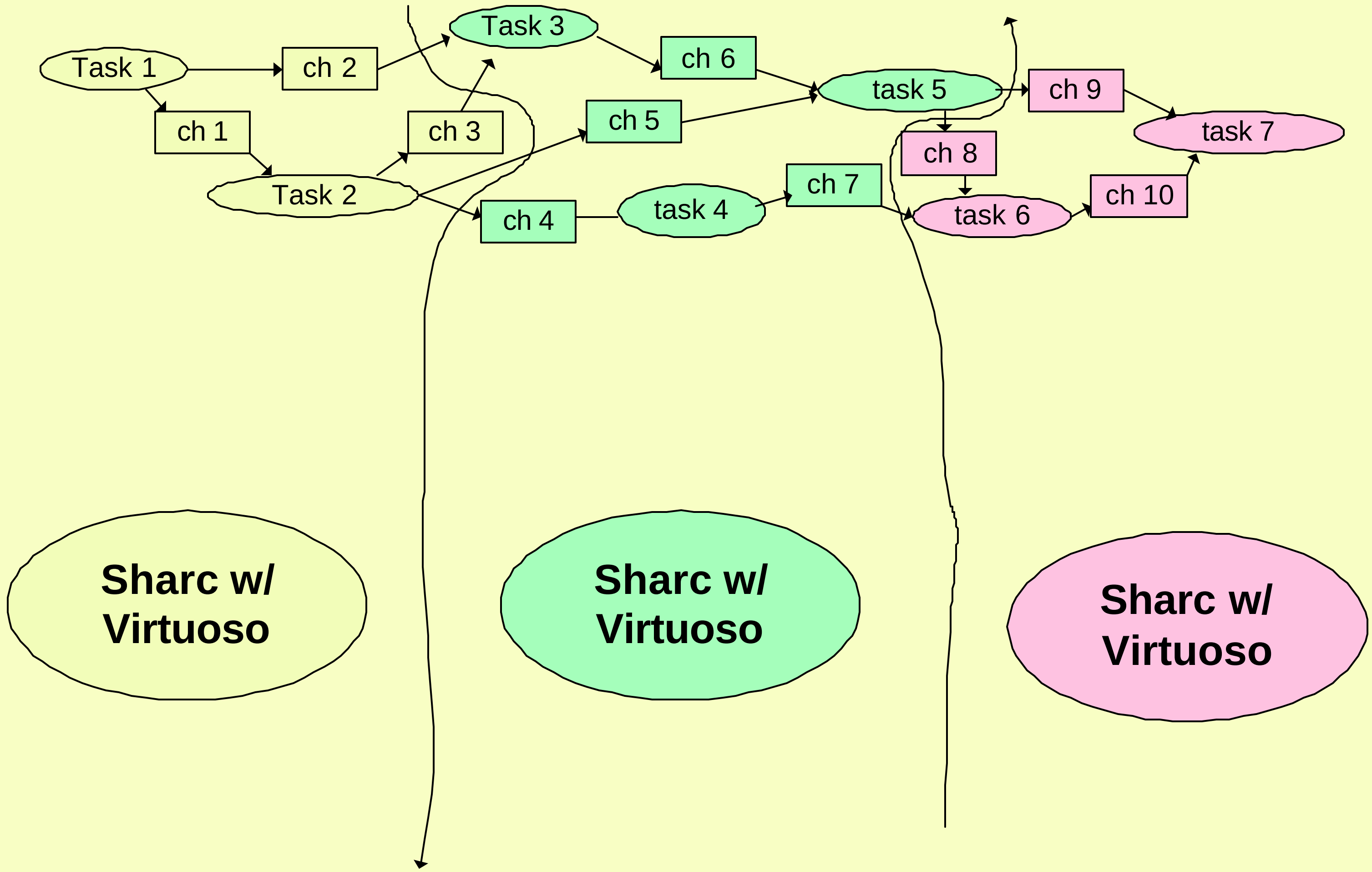
Full application : Matlab/Simulink type design

- Embedded DSP app with GUI front-end

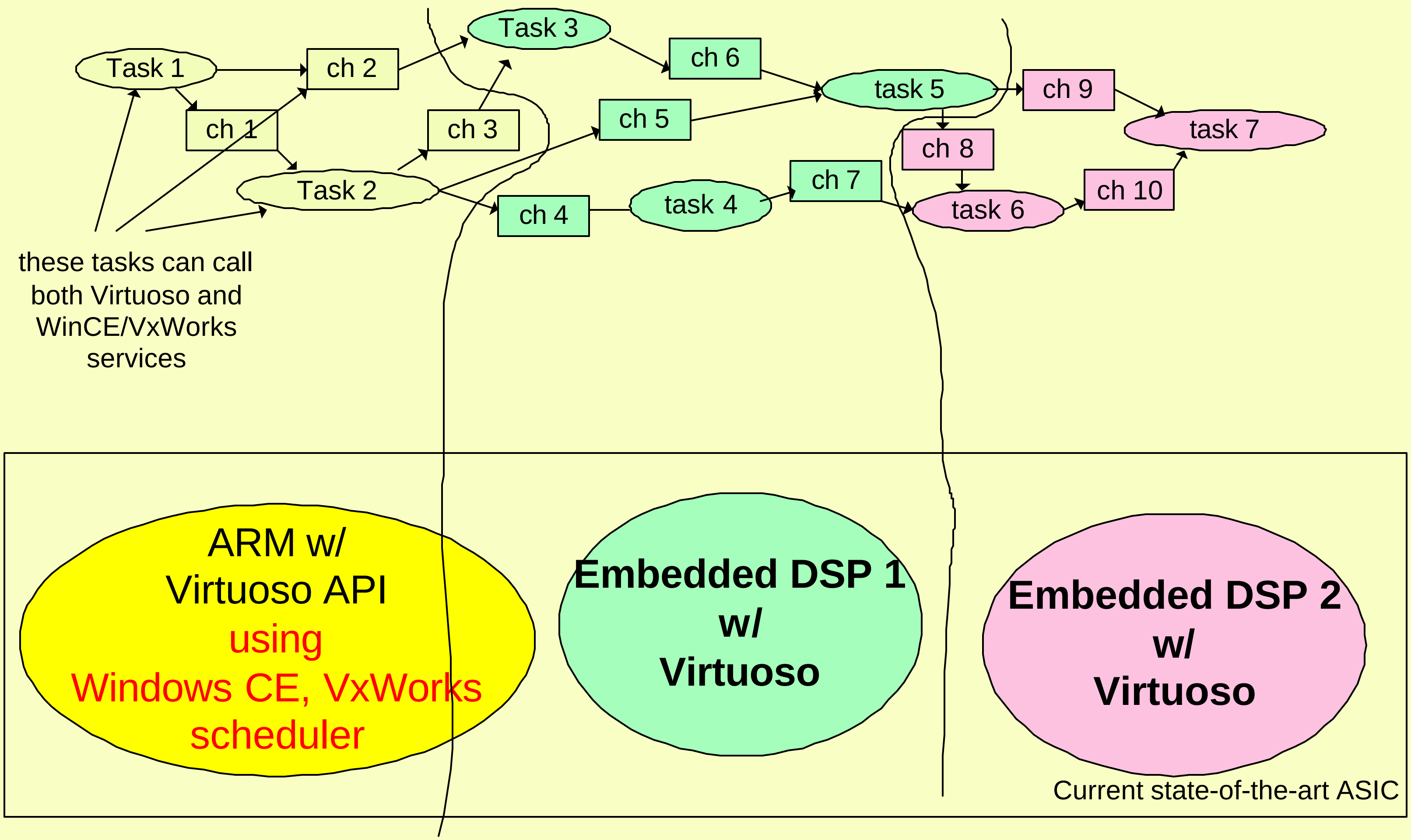


Virtuoso VSP off-the-shelf

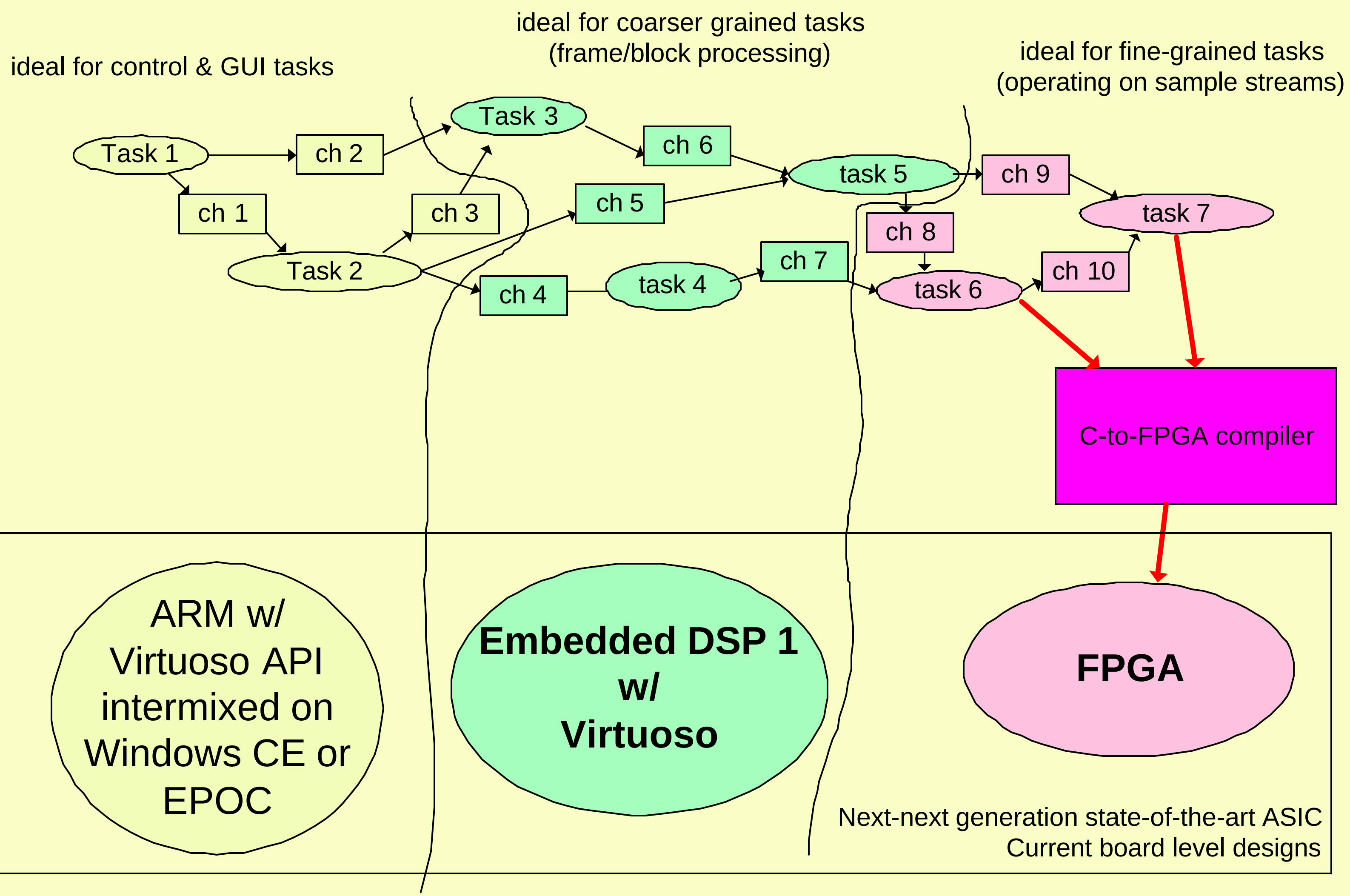
Block diagram at top level, executable spec in e.g. C



Today : Heterogeneous VSP with host OS



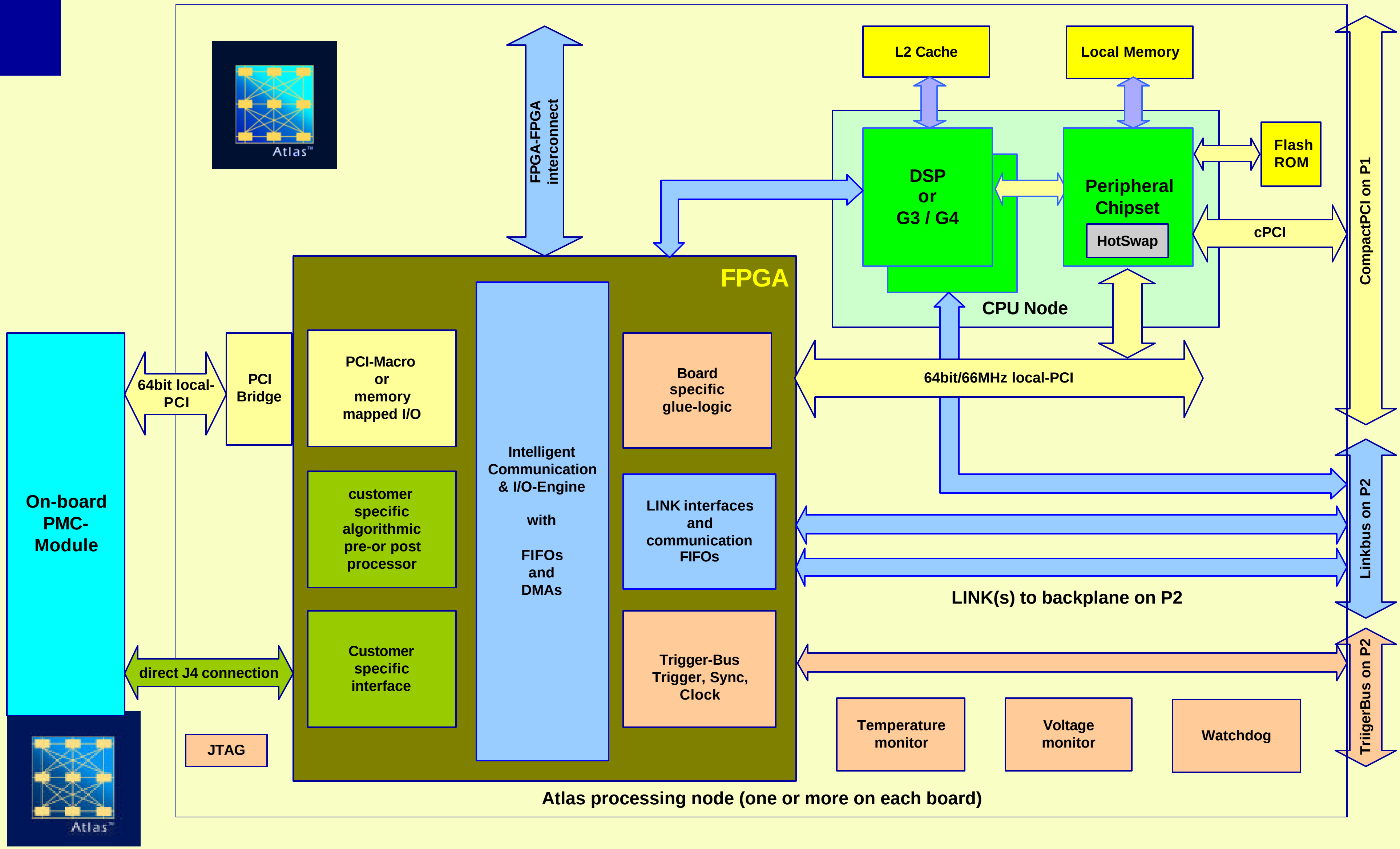
Heterogeneous VSP with reprogrammable HW



Eonic's CSPA concept : board level architecture

- CSPA : Communicating Signal Processing Architecture
- Designed for high-end scalable DSP systems
- Central ideas :
  - Scalability (up or down) from 1 to 1000's of processors
  - Distribute everything : I/O, processing, communication
  - Hence, link based communication (bus is slow I/O device)
  - "Active communication backbone" : by using FPGA
  - Must be supported by software programming model
- Result :
  - Very scalable
  - No bottleneck for processing : can be done in communication stream
- Problems found :
  - Many processors lack busses and DMA
  - Bus bridges and interfaces become too complex (if it works at all)

CSPA as implemented on Eonic's Atlas



Links and switch fabrics

- Links : idea pioneered by INMOS transputer, putting CSP model in HW
- Switch fabrics : as busses are hitting the wall, “switch fabrics” are called at the resque. Especially for broadband telecom
- But : why do switch fabrics like RapidIO, Infiniband, etc. have support for e.g. “cache coherency in shared memory ?, PCI interfaces ?
- Reason : programming model and architectural assumptions kept unchanged
- But : how to handle 12+ wires, each at Gbit/s that have to keep in sync ?
- What happens when such signals go off-chip, go through PCB, connectors, backplane, … ?
- Needed : go bit serial with LVDS type signaling, clock recovery from data, 8/10 bit encoding, DMA, FIFO, flow control, runtime error detection and recovery, hot reconnect, remote reset
- Solutions : back to basics = simple, but complete and flexible
- Example : IEEE1355, Spacewire : just a link with higher level protocol
- Result : less gates, less special circuits, less power, better performance and RELIABILITY !

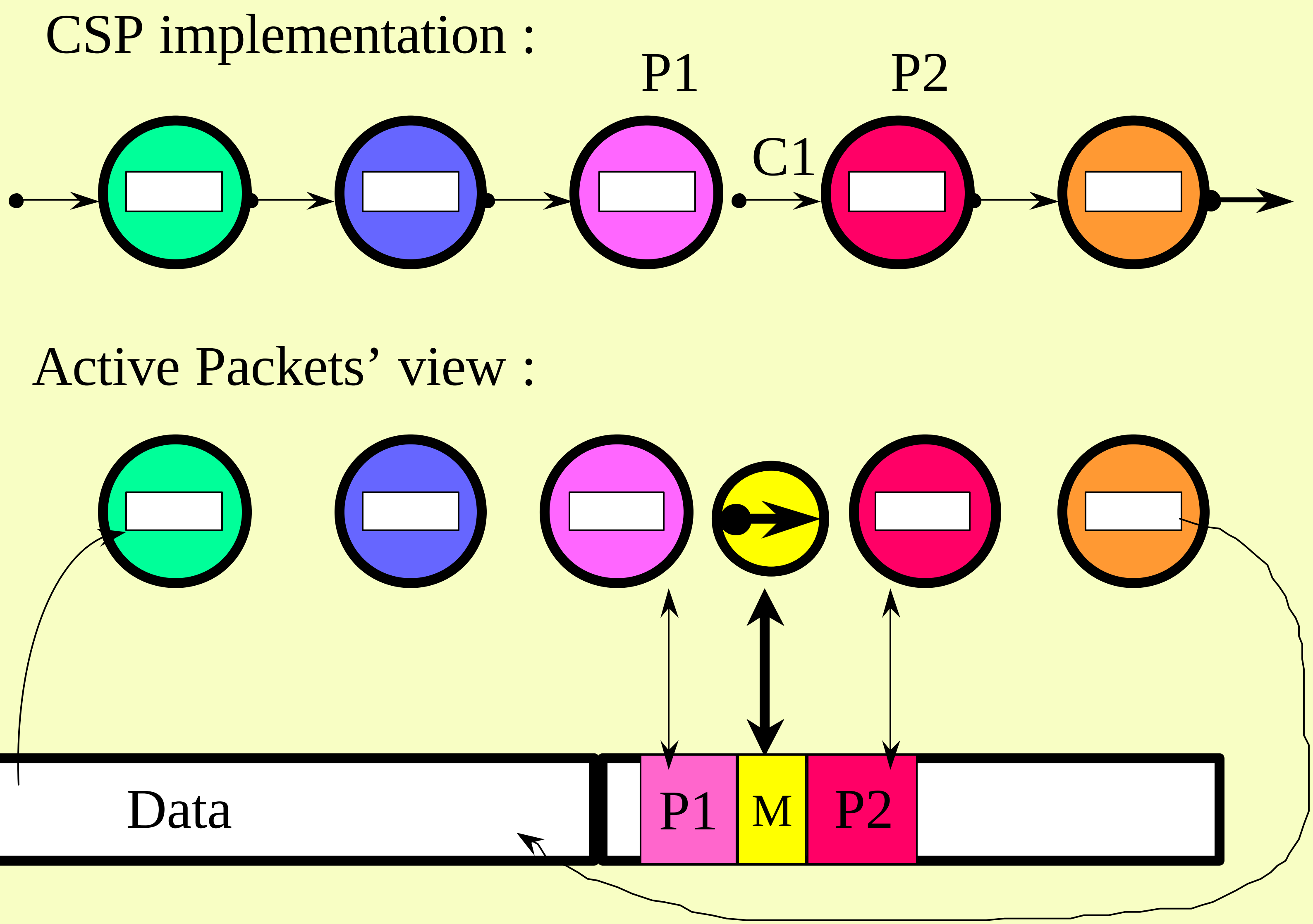
Beyond multi-tasking

- The CSP model acts as a hierarchical compositor for sequential (procedural) processes
- Problem is now how to handle the “connections” and the communication protocols
- Hence : statically defined programs
- Problem domains :
  - runtime changes
  - I/O and memory management become explicit
  - Programming languages reflect control flow architecture of original Von Neumann machine

From procedure to data oriented

- Today's procedural view :
  – Output = F (input)
  – F is central
  – input and output is peripheral activity
  – Time introduced as a side-effect and a buffer
- Another view : merge data and procedures -> functional view
  – $[\text{Data*(F\_output)}]_{t+n} = [\text{Data(F)}]_{t}$: DSP natural !
  – procedures and data are bundled into "active" packets
  – runtime loading and scheduling allows for self scaling and resilience to errors, makes it time-neutral

CSP & Active Packets



Conclusion

- RTOS is much more than real-time
- General purpose “process oriented” design and programming
- Hide complexity inside chip for hardware (in SoC chip)
- Hide complexity inside task for software (with RTOS)
- Hide complexity of communication in system level support
- CSP provides unified theoretical base for hardware and software, RTOS makes it pragmatic for real world :
  – “DESIGN PARALLEL, OPTIMIZE SEQUENTIALLY”
- Software meets hardware with same development paradigm :
  – Handel-C for FPGA, “Parallel” C for SW
- FPGA with macro-blocks is pre-cursor of next generation SW defined SoC :
  – Needs concurrent SW development paradigm
  – Needs standardized communication backbone
- Time for asynchronous HW design ?