COMPSYS 701: Advanced Digital Design

Group and individual (research) projects (GP and IP)

Designing a Heterogeneous Network-on-Chip Multiprocessor for Mixed-criticality Applications

Zoran Salcic

Department of Electrical, Computer, and Software Engineering

Semester 1, 2025, University of Auckland

Motivation and background

Multi-core systems on chip (SoC) are capable of achieving both high (average) and guaranteed absolute performance (response times, energy consumption) with some challenges to be addressed and resolved:

- How to use different types of cores and their combinations to effectively achieve target goals of the applications
- What type of interconnect/NoC is suitable for certain types of applications
- How to guarantee properties and performance of certain parts of applications and time-predictability of the execution platform when dealing with hard real-time systems/applications
- How to separate non-critical and critical parts of an application on the same execution platform and work unison

Motivation and Background

- Heterogeneous multi-core systems comprise of different types of cores, e.g.:
  - Traditional processor cores (typically RISC type)
  - Customised processors that emphasise specific architectural targets (e.g. GPUs)
  - Application specific processors
  - Various accelerators
  - Interconnect mechanisms of different performance levels
- As the whole, when implemented in FPGAs allow full customisation (number and type of cores, interconnect)
- Can be optimized for one or more design criteria (speed/throughput, real-time, energy consumption, criticality, etc)

Motivation and Background

Issues in the design of heterogeneous MPSoCs:

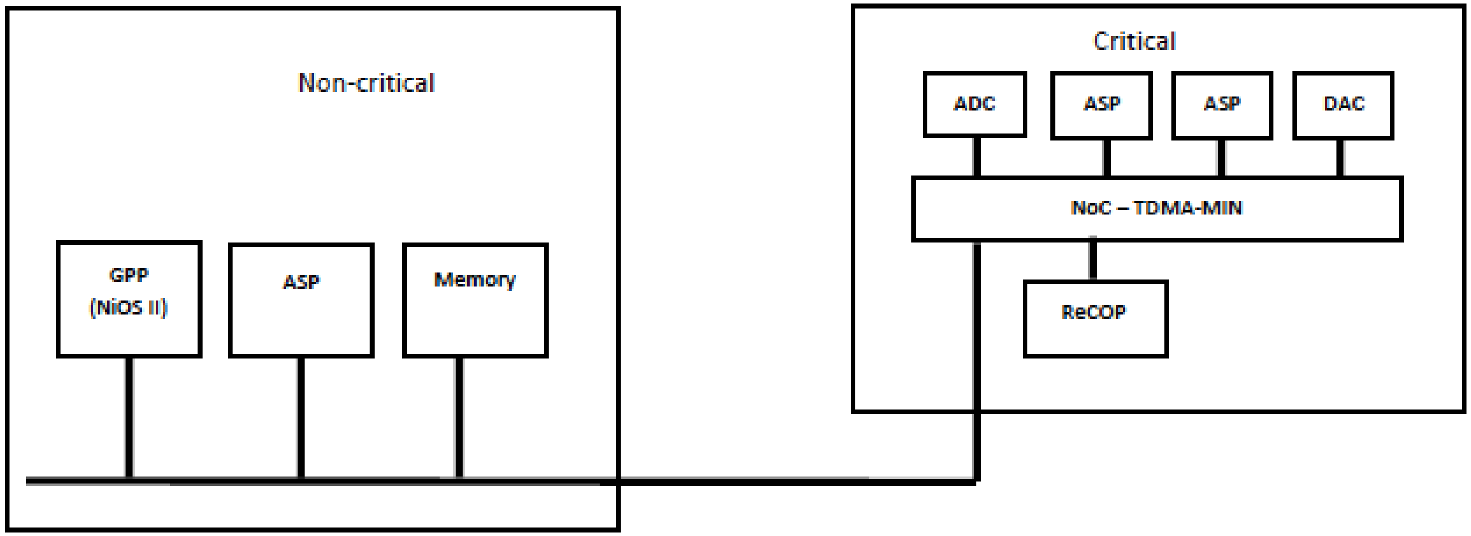
- How to specify the application (design/programming language(s))
- How to allocate tasks/functions and schedule their execution
- How to analyse and guarantee certain features/performance parameters
- How to use/perform HW/SW partitioning
- What strategy to apply for different design criteria
- How to address mixed control/data-dominated applications with (hard) real-time requirements

Our overarching and long-term goal:

To make major elements of execution platform for executing programs specified/designed using **SystemGALS** language based on formal Model of Computation (MoC) which is friendly with standard programming/design languages

Heterogeneous MultiProcessor System-on-Chip - HMPSoC

- ReCOP processors for control-dominated parts
- ASPs for data-dominated parts and
- TDMA-MIN for internal physical communication in critical part
- NiOS 2 for non-critical parts

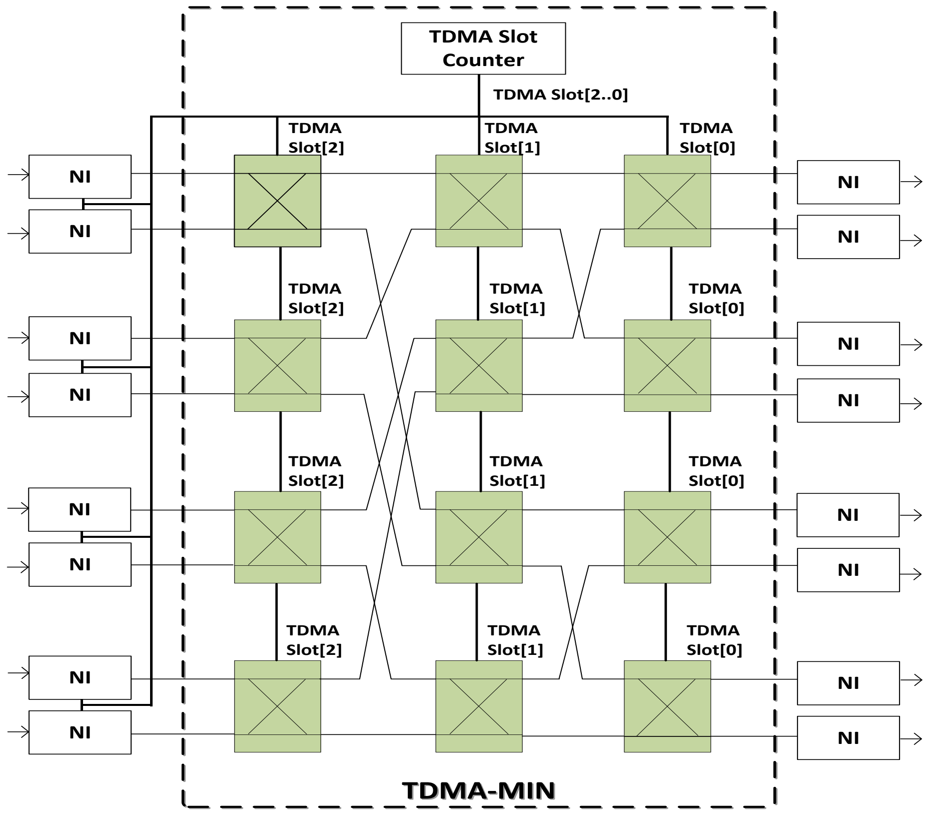


Group and Individual Project Goals

Targets Heterogeneous Multi-Processor/core System on Chip (HMPSoC) and specific design flow (not yet automated):

- New specification language SystemGALS (GALS Model of Computation)
- Initially targets FPGA-implemented NoC-based MPSoCs
- Uses **ReCOP** reactive processor for control-dominated parts in time-critical applications, as well as event-driven functions
- Data-dominated parts are platform-agnostic (can be any processor, multi-core system or application-specific processors)
- Data-dominated parts with very clear and strict interface can be specified/designed in “any” language (e.g. C, Java, VHDL etc)
- In this project data-dominated parts are using Application-Specific Processors (**ASP**s), but can also use any General Purpose Processor (NiOS 2 in our case)
- NoC – Time-Division-Multiple Access Multi-Stage Interconnect **TDMA-MIN**

Time-Division Multiple Access – Multistage Interconnect Network (TDMA-MIN)



- Allows transfer of data between any source and any destination node without conflicts with other simultaneous transfers with bounded transfer time
- Can implement TDMA rounds that guarantee transfers between source and destination node in every round
- Fully time-predictable
- TDMA-round consists of multiple time slots

TDMA-MIN Interconnect

- Given N I/O ports and nodes in network, there are s transmission stages where

$$s = log_2 N$$

- Each stage features N_s 2x2 crossbar switches where

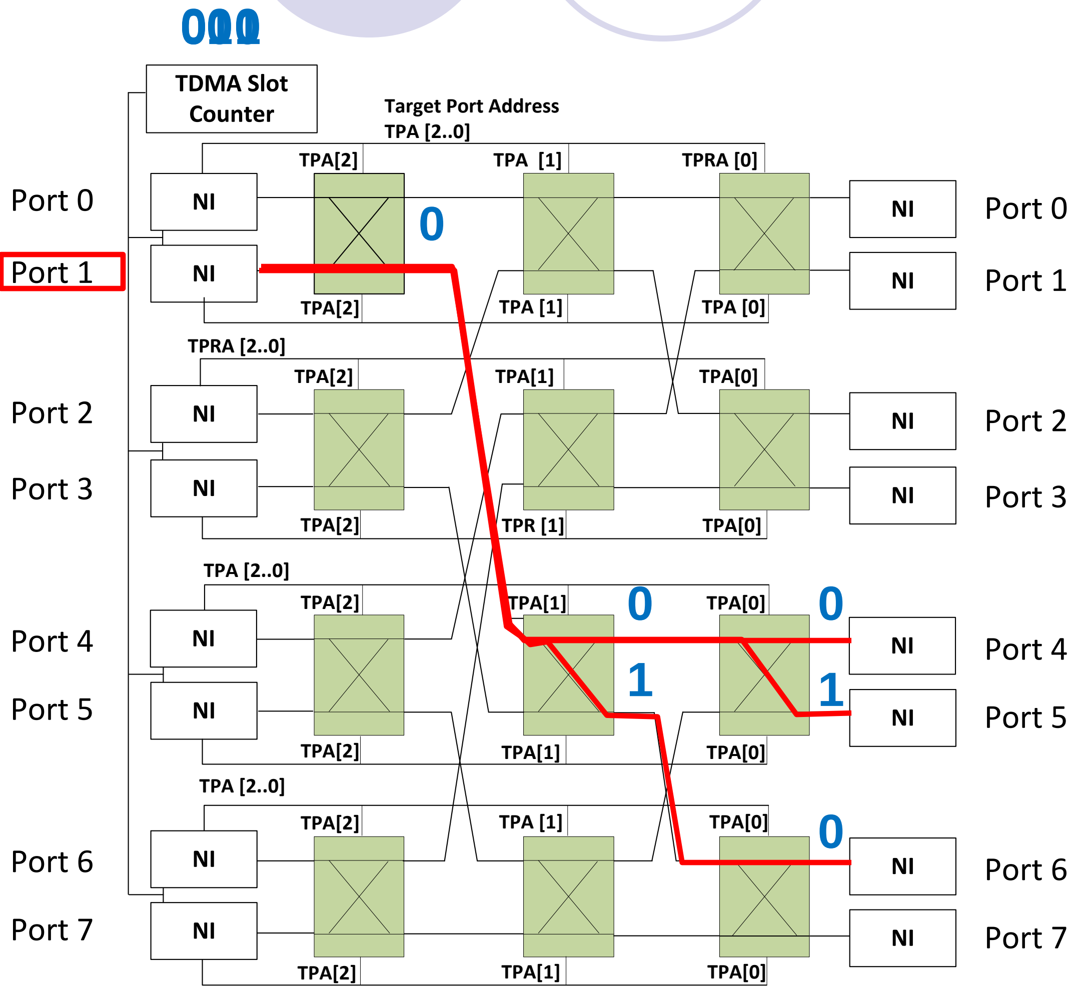
$$N_s = N/2$$

- A counter fed into the network interface guarantees no packet conflicts
  - Connections between input and destination ports are calculated by:

$$n_{rx} = Mirror(n_{tx}) \oplus T_i, \qquad where\ i \in \{0,1,\ldots,N-1\}$$

n_{rx} - receiving port (binary address)
n_{tx} - transmitting port (binary address)
T_i - TDMA slot (binary address)

TDMA-MIN Interconnect for 8 Ports



- ➢ A slot counter controls all crossbar switches in the network
- ➢ 3-bits for 3-stage networks
  - 0 – Parallel
  - 1 – Cross

$$s = log_2 8 = 3\ stages$$

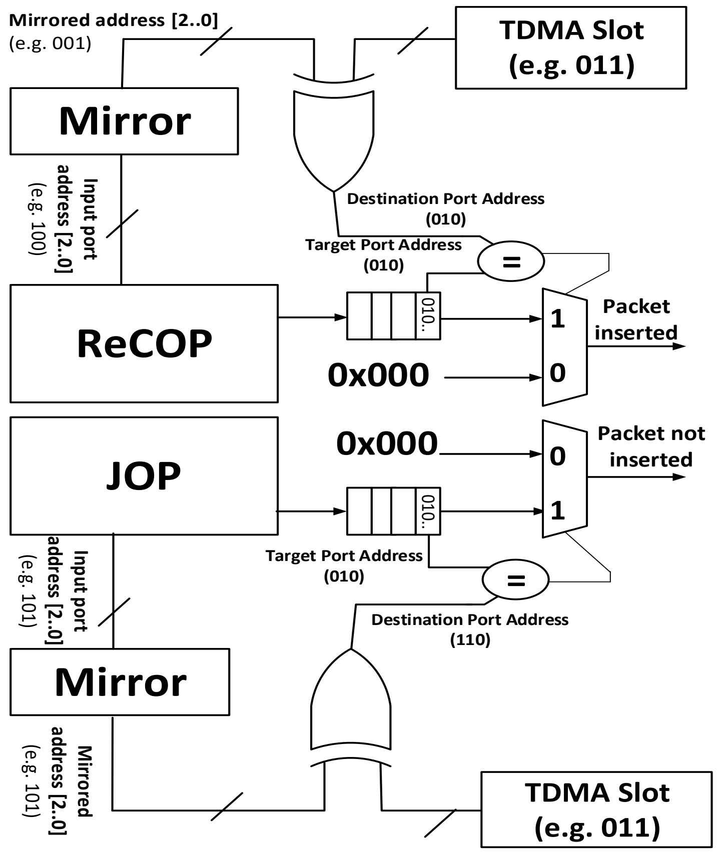
$$N_s = \frac{8}{2} = 4\ switches\ in\ each\ stage$$

No conflicts at any given slot

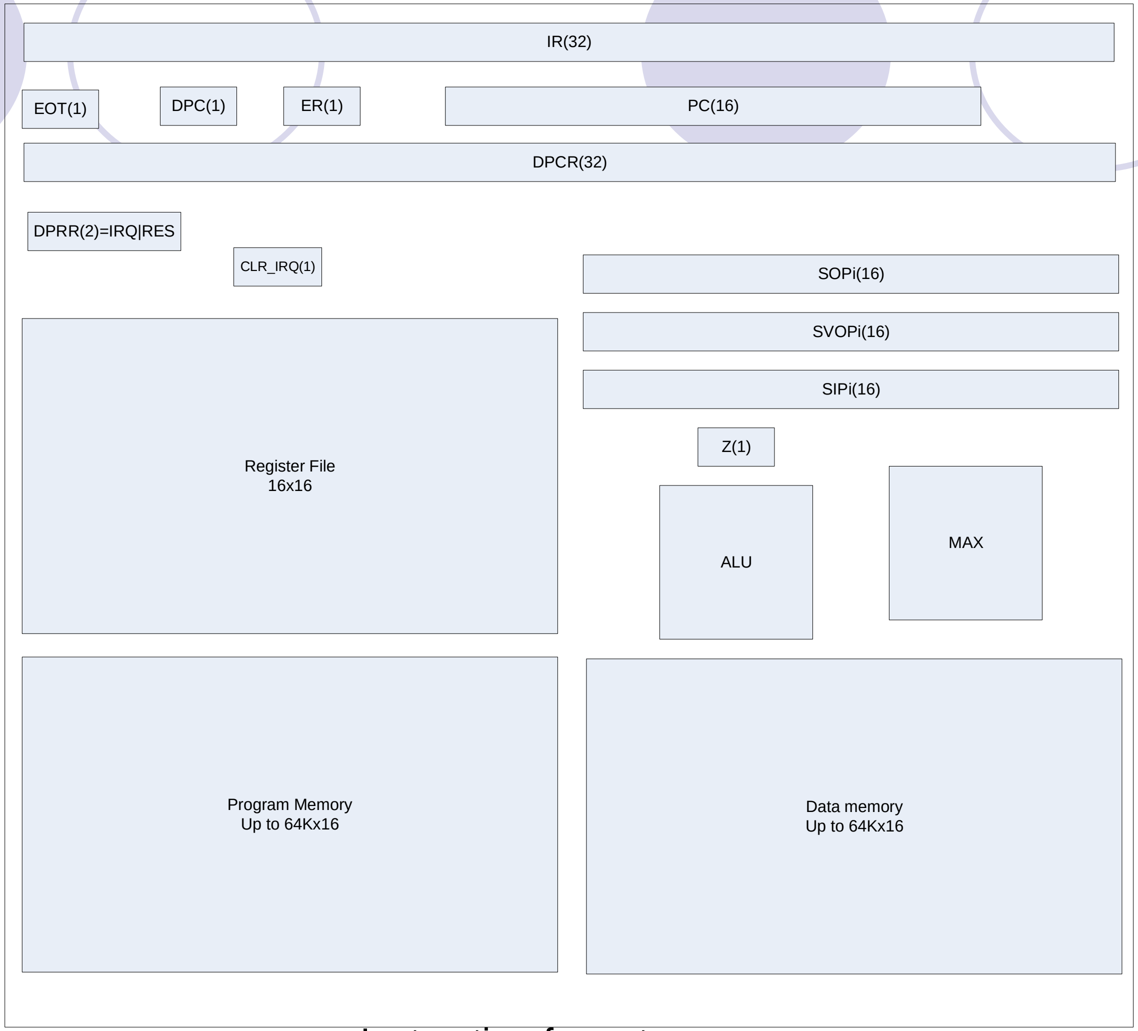
Input port	Mirror	Destination port in TDMA slots							
		slot 0	slot 1	slot 2	slot 3	slot 4	slot 5	slot 6	slot 7
0	0	0	1	2	3	4	5	6	7
1	4	4	5	6	7	0	1	2	3
2	2	2	3	0	1	6	7	4	5
3	6	6	7	4	5	2	3	0	1
4	1	1	0	3	2	5	4	7	6
5	5	5	4	7	4	1	0	3	2
6	3	3	2	1	0	7	6	5	4
7	7	7	6	5	4	3	2	1	0

Packet insertion mechanism

- Calculated the address of the destination port
- Mirror the port number of the processor making datacall
- Destination port address obtained by XORing the mirrored value with TDMA Slot #.
- The datacall (packet) is read from the FIFO and target address provided in the datacall is compared against the calculated destination port address.
- The packet will be transmitted through the network if matches occur during the current TDMA slot



ReCOP - Programming Model

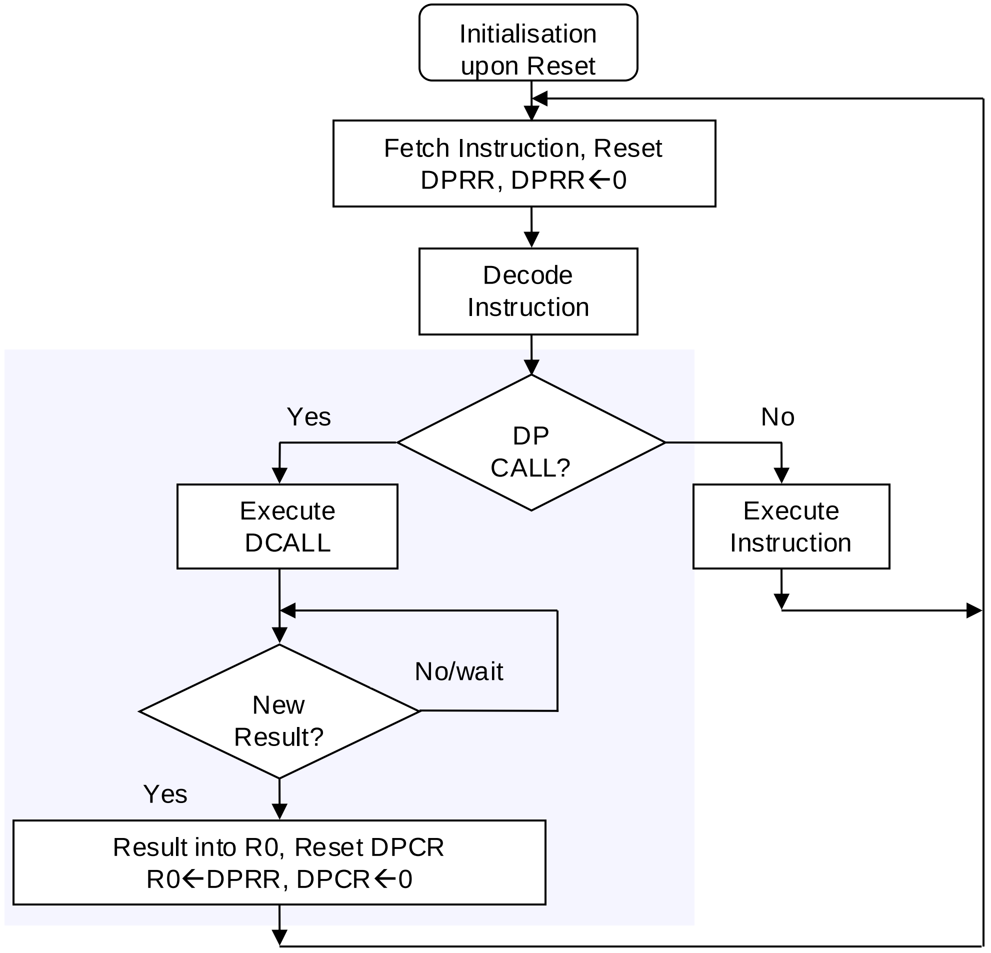


Instruction format

AM (2)	Opcode(6)	Rz(4)	Rx(4)	Operand(16)

ReCOP – Reactivity and Concurrency Processor

Control-unit operation



Initialisation upon Reset
Fetch Instruction, Reset DPRR, DPRR←0
Decode Instruction
DP CALL?
Yes
No
Execute DCALL
Execute Instruction
No/wait
New Result?
Yes
Result into R0, Reset DPCR R0←DPRR, DPCR←0

What if non-blocking call?

ReCOP – Instruction Set

Instruction	Description	Register Transfers	Addressing Modes			
			Inherent	Immediate	Direct	Register
AND Rz Rx #Operand	The contents of Rx and Rz / Operand are ANDed and the result is stored in Rz	Rz <- Rx AND Operand		X		
AND Rz Rz Rx		Rz <- Rz AND Rx				X
OR Rz Rx #Operand	The contents of Rx and Rz / Operand are ORed and the result is stored in Rz	Rz <- Rx OR Operand		X		
OR Rz Rz Rx		Rz <- Rz OR Rx				X
ADD Rz Rx #Operand	The contents of Rx and Rz / Operand are added and the result is stored in Rz	Rz <- Rx + Operand		X		
ADD Rz Rz Rx		Rz <- Rz + Rx				X
SUBV Rz Rx #Operand	The contents of Rx and Rz / Operand are subtracted and the result is stored in Rz	Rz <- Rx - Operand		X		
SUB Rz #Operand	The contents of Rz and the Operand are subtracted but the result is not stored	Rz - Operand		X		
LDR Rz #Operand	Load Rz with the content of immediate value / memory location pointed to by Rx or Operand	Rz <- Operand		X		
LDR Rz Rx		Rz <- M[Rx]				X
LDR Rz $Operand		Rz <- M[Operand]			X	
STR Rz #Operand	Store the content of Rx / immediate value, into memory location pointed to by Rz / direct address	M[Rz] <- Operand		X		
STR Rz Rx		M[Rz] <- Rx				X
STR Rx $Operand		M[Operand] <- Rx			X	
JMP #Operand	Jump to address location unconditionally	PC <- Operand		X		
JMP Rx		PC <- Rx				X

ReCOP – Instruction Set

Instructions in red font are not required in implementation, but additional instructions may be required depending on project team's decisions

Instruction	Description	Register Transfers	Addressing Modes			
			Inherent	Immediate	Direct	Register
PRESENT Rz #Operand	Jump to address location if the thread pointed to by Rz is not present else continue execution	if Rz(15..0)=0x0000 then PC<-Operand else NEXT		X		
DATACALL Rx	Store the content of R7 and Rx in DPCR register	DPCR <- Rx & R7				X
DATACALL Rx #Operand	Store the content of Rx and Operand in DPCR register	DPCR <- Rx & Operand		X		
SZ Operand	Jump to address location if Z=1 else continue execution	if Z=1 then PC <- Operand else NEXT		X		
CLFZ	Clear Zero flag	Z <- 0	X			
CER	Clear ERready bit	ER <- 0	X			
CEOT	Clear EOT bit	EOT <- 0	X			
SEOT	Set EOT bit	EOT <- 1	X			
LER Rz	The content of ER is stored in Rz	Rz <- ER				X
SSVOP Rx	Load SVOP with the content of Rx	SVOP <- Rx				X
LSIP Rz	Load Rz with the content of SIP	Rz <- SIP				X
SSOP Rx	Load SOP with the content of Rx	SOP <- Rx				X
NOOP	No operation		X			
MAX Rz #Operand	Comparing two 16-bit and stores larger one into Rz	Rz <- MAX{ Rz, #Operand}		X		
STRPC $Operand	Stores program counter into the specified memory address	M[Operand] <- PC			X	

Group Project 1 – GP1 – Implementation of ReCOP

- Goal: Full ReCOP implementation in the FPGA (Cyclone V) of DE1-SoC
- Define final blocks of the datapath and instruction set as needed for project
- Define full programming model that includes all interface registers and memories and decisions regarding memories
- Define mechanism for storing a program in program memory without resynthesis of the processor (by changing PM content through .MIF file or by external control/source)
- In parallel develop a simple Assembler (define source instruction syntax/format, labels, program address start and anything that help programmers to efficiently use ReCOP from Assembly language)
- Initial instruction set given (without instructions in red font)

Group Project 1 – GP1 – Implementation of ReCOP

- Specify the datapath with full external interface including DE1SoC peripherals (e.g. 7-segment display, push buttons, switches, LEDs) and connections with Control Unit; also plan interconnection to NoC
- Specify the Control Unit (CU) operation and its external interface (datapath control signals, other/external control signals, status signals, other/external inputs); assume multicycle implementation (no pipelining)
- Develop ModelSim model and simulate integrated processor within a testbench Program and data memories can be part of testbench); extract the part for synthesis

Group Project 1 – GP1 – Implementation of ReCOP

- Synthesise the model and develop ReCOP IP with the well(fully)-defined external interface
- Demonstrate operation of ReCOP when executing example program(s)
- Make clear plan on extensions that will be added in GP2 when integrating ReCOP within HMPSoC
- Write a short report that explains additions and customization compared with original processor
- Complete project resources (codes, design, run procedure, readme file etc) for submission

Individual Project (IP) - Application-Specific Processor (ASP)

- Agree within group who will be doing what as your IP (block) target (make individual assignment)
- Each block developed as an ASP acts as a node connected to NoC
- Some ASPs are specialised for input/output (IO-ASP) including to analog world (Act as advanced Analog-to-Digital/Digital-to-Analog, ADC/DAC, ADC-ASP, DAC-ASP)
- IO-ASP (Example-ASP in Reference Design) with configurable parameters (e.g. sampling frequency, down-sampling/up-sampling rates, channel selection, mode delivery samples (single/block) etc) selectable by the external command issued from ReCOPs or other ASPs)
- Computation/data processing ASPs (DP-ASP) that perform complex computations more efficiently and are customised

IP- Application-Specific Processor (DP-ASP)

- A data processing, DP-ASP, can execute certain functions much more efficiently than general purpose processor (because of HW implemented functions)
- In HMPSoC, they can create operational “pipelines” and act as specialist “workstations” that perform operations such as
  - Direct passthrough of data stream
  - Averaging of incoming data stream
  - Auto-Correlation
  - Peak Detection
  - Source data generator (emulates physical input signals with their digital counterparts)
- Other more advanced operations/algorithms such as fast search, sorting, pattern recognition, NN inference etc
- Non-critical part with Nios 2 supports critical part; also used in debugging, changes of software version in critical part, global configuration
- Nios 2 can be customized to support some specific operations through new instructions

IP - Application-Specific Processor (ASP)

- DP-ASP can provide certain operations as services requested by either control-flow or data-flow, where there is a need for clearly specified and “standardized” interfaces and communication protocols, as well as convention for data storage
- Investigation of functionality and prototype implementation of a few ASPs is a goal of **Individual (Research) Project**
- Target application is fixed (power system frequency measurement) In addition to the main function of each ASP, the attention needs to be paid to interfacing of the ASP with the TDMA-MIN
- More details on IP and ASPs will be given in separate lecture

Group Project 2 (GP2) - HMPSoC and an Example Application

- A HMPSoC will be designed and synthesized with a single NiOS, single ReCOP and four ASPs connected through TDMA-MIN NoC as a platform that will run an application
- We envisage an application that will have a few (at least 2) modes of operation controlled within control-flow (control-dominated part) which can be invoked depending on events coming from external environment (or generated by human operators)
- Nios can be the execution platform for providing details of control flow, and ReCOP as the executor of control flow and processor that configures selected ASPs in HMPSoC and establishes data-flow between ASPs. A simple interface on DE1-SoC can be used to initiate instantaneous actions of ReCOP, e.g. change of configuration and functionality of ASPs and modification of processing pipeline.
- The application we are targeting will enable testing of HMPSoC in realistic scenario of **real-time measurement of power network frequency**
- The application can be be specified in pseudo SystemGALS language code or some other suitable specification mechanism
- Note: ReCOP has only an Assemby language, no high-level languages

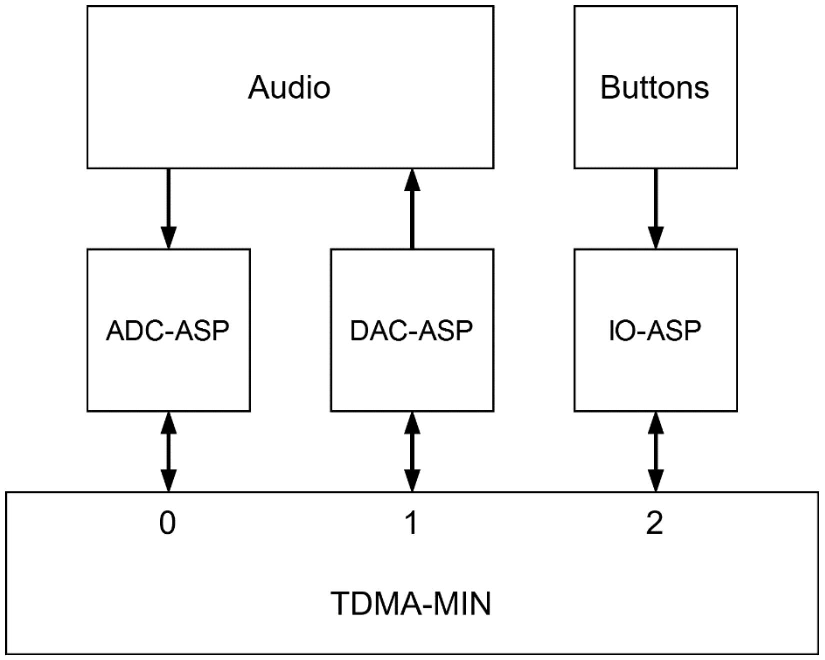
Reference Design for a System Nucleus

- Familiarisation with the NoC-based approach will be done in Lab 2 through the example called “NoC Reference Design”
- Three application specific nodes connected through the NoC
  - ADC-ASP, accepts data stream from audio signal source (Audio Module) and splits it into two streams corresponding to Left and Right channel; it forwards the stream to the IO-ASP
  - IO-ASP/Example-ASP, receives commands from the push buttons (keys) to ”process” audio signal (mutes or just forwards signal to DAC-ASP); it also configures all ASP nodes in the system
  - DAC-ASP, receives the data streams (L and R) from IO-ASP and forwards it to Audio Module for listening over headphones
- Initial configuration message and data message formats are provided and can be changed

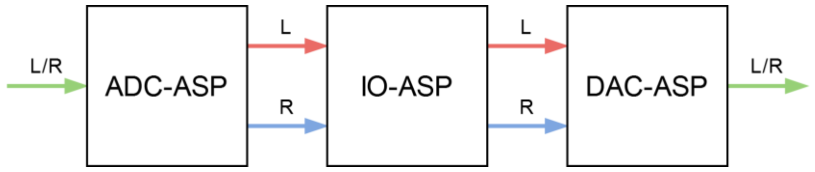
Reference Design (RD) as Critical Part Nucleus

Implemented RD platform

(IO-ASP also called Example-ASP)



Data flow



Reference Design for a System Nucleus

	Bits																						
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	...	0
Data-Audio	1	0	0	0	Dest				Reserved							Ch	16-bit signed integer							
Conf-DP	1	0	0	1	Dest				Next				Mode				Unused							
Conf-ADC	1	0	1	0	Dest				Next				SR		En	Ch	Unused							
Conf-DAC	1	0	1	1	Dest				Reserved				SR		En	Ch	Unused							

- ADC-ASP and DAC-ASP can be retained as the initial design for the HMPSoC, first for critical part of the system and then used as a component in Platform Designer to connect with Nios II

Major Tasks in Project Implementation

- Make ReCOP Design (multi-cycle) with customized interfaces
- Write a simple Assembler that translates assembly code into the object code
- Demonstrate ReCOP operation on DE1-SoC board by running meaningful small programs translated using your Assembler
- Implement programs on ReCOP for configuration and change of configuration of other nodes and invocation of operations on DP-ASP
- Plan interfaces of DP-ASP (from IP) and ReCOP to connect to NoC
- Familiarize and try Reference Design
- Implement the functionality of Reference Design with ReCOP instead of IO-ASP
- Plan common part of DP-ASP with other members of group

Major Tasks in Project Implementation

- Plan the role of Nios II in HMPSoC (e.g. changing ReCOP program with a direct access to ReCOP program memory, receiving data for debugging and visualization from other nodes, other non-real-time functions, running control flow part of SystemGALS program)
- Include custom instructions into Nios ISA (e.g. to control of Critical Part nodes, operations)
- Design ReCOP program reconfiguration node that enables reconfiguration of critical part
- Update ReCOP assembler if needed (especially if you add new instructions to ReCOP)
- As part of IP each student has to design additional function(s) of their choice (from the function specified in IP brief) as an ASP with necessary interfacing (Network Interface to NoC, and parameterization of functions)
- More constraints on IP topic as part of the group's decision making with power system frequency measurement
- (DP-ASPs are implementing DMs of SystemGALS; Nios functions can also be SW-implemented DMs)

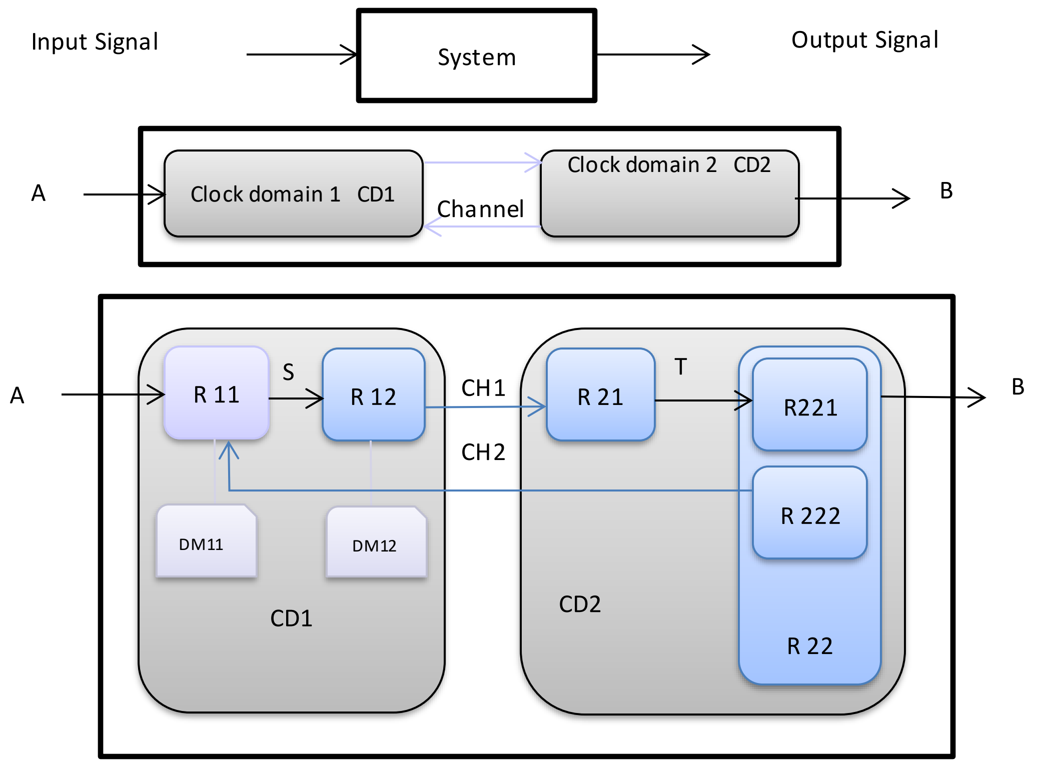
SystemGALS

A GALS language for specification of GALS systems:

- Uses GALS MoC as underlying system design concept
- Control-dominated part (almost) identical to SystemJ
- A system consists of asynchronous design units, clock domains and within them of synchronous design units, reactions
- Communication within clock domains using signal object and between clock domains channel as rendezvous (with full handshaking)
- Data dominated-part in the form of concentrated “data modules”
- Control flow variables (CFVs) introduced to enable synchronization between control flow and data modules

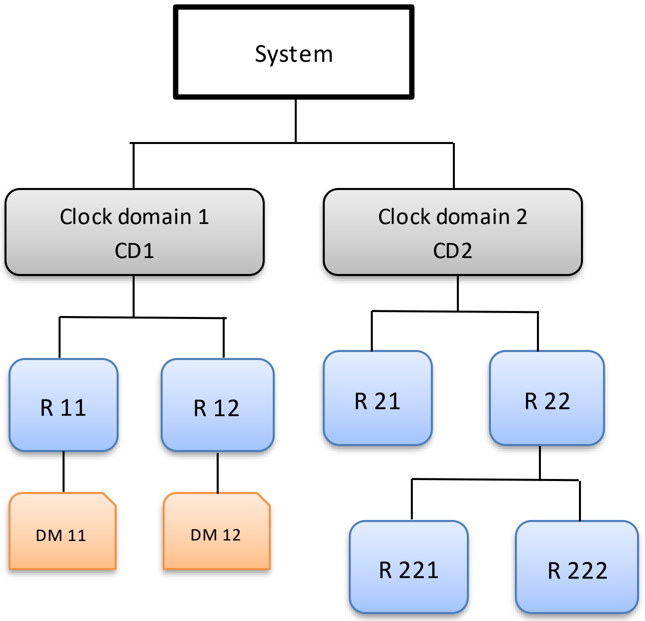
SystemGALS – Graphical representation of GALS System

SystemGALS system visualized with graphical representation:



SystemGALS – Graphical representation of GALS System

SystemGALS system hierarchical representation:



SystemGALS – Synchronous Kernel Constructs

Synchronous Kernel Constructs

#	**Kernel constructs**	**Meaning**
1	`p1;p2`	Sequential execution/composition of p1 and p2
2	**`pause`**	Completing a tick
3	`[`**`input`**`|`**`output`**`] [type]` **`signal`** `S`	Declaring an interface (input or output) or a local signal with a Boolean status and an optional value
	`cfv` `V`	Declaring a control flow variable V
4	**`emit`** `S`	Emitting signal S: sets signal status to present, optionally set a value if given as an expression
5	**`while`**`(true) p`	Temporal loop: every iteration consumes a single tick (p must contain at least one pause statement)
6	**`present`**`(Sexp`$^{1)}$`) {p1}` **`else`** `{p2}`	Execute p1 if signal S or signal name expression is set to present in the previous tick, otherwise execute p2
7	**`if`** `(CFVexp`$^{2)}$`) {p1}` **`else`** `{p2}`	Executes p1 if CFVexpr value is true, otherwise execute p2

SystemGALS – Synchronous Kernel Constructs

#	Kernel constructs	Meaning
8	**if** (CFVexp2)) {p1} **else** {p2}	Executes p1 if CFVexpr value is true, otherwise execute p2
9	[**weak**] **abort**([**immediate**] Sexp) {p}[**do** {q}]	Preempts p if S was present in the previous tick
10	[**weak**] **suspend**([**immediate**] Sexp) {p}	Suspends execution for a tick if S was set present in the previous tick
11	**trap**(T) {p}	Software exception
12	**exit** T	Throws a software exception
13	p1 || p2	Combining p1 and p2 to execute in parallel synchronously
14	{+ [stmts] +}	Data module inline call developed in host language

SystemGALS – Asynchronous Kernel Constructs

Asynchronous Kernel Constructs		
15	**output** type **channel** C	Sending channel declaration
16	**input** type **channel** C	Receiving channel declaration
18	**send** C	Sends data, exp, through channel C
19	**receive** C	Receives data through channel C

SystemGALS – Data Modules and Interface Functions

DataModule func(arguments[3)]){+ [stmts] +}	Declaration of data module
dmcall func(arguments[3)])	Call of a data module within reaction
sgl_GetSigVal[4)]	Function that extracts the signal value for the data computation
sgl_SetSigVal[4)]	function that assigns the signal value from the data computation
sgl_GetSigStatus	Function that returns signal status
sgl_GetChanVal[4)]	function that extracts the channel value for the data computation
sgl_SetChanVal[4)]	function that assigns the channel value from the data computation
sgl_SetCFV	sets the value of CFV to 0, 1 [false, true]
Sgl_GetCFV	Function that extracts CFV status

SystemGALS – In-line Declared Data Modules

```
CD(input int signal A; output int signal B;)->{
 await(A); // waiting for A
 {+ // beginning of the data module
  int result;
  int val = *(int *)sgl_GetSigVal(A);  // getting the value
  // perform computation based on the retrieved value
  result = userDefinedFunction(val);
  sgl_SetSigVal(B, &result, sizeof(result)); // setting the value
 +} // end of data module
 emit B; // emitting B with a value 'result'
}
```

SystemGALS – Interfacing Control Flow with the Data Module

```
CD(input int signal A; output signal B,C;)->{
 cfv result = false; // declaring a cfv
 await(A); // waiting for A
 {+
  int val = userDefinedFunction(getSignalValue(A));
  if (val > 20)
   sgl_SetCFV(result, 1); // 1 : true
  else
   sgl_SetCFV(result, 0); // 0 : false
 +}
 // emitting a signal based on the cvf 'result'
 if(result)
  emit B;
 else
  emit C;
}
```

SystemGALS - Data Module Function

```
DataModule func1(signal arg0, signal arg1){+
 int result;
 int val = sgl_GetSigVal(arg0);
 result = userDefinedFunction(val);
 sgl_SetSigVal(arg1, &result, sizeof(result));
+}

DataModule func2(Signal arg0, cfv arg1);

CD(input int signal A; output int signal B;)->{
 cfv result = 0;
 await(A); // waiting for A
 dmcall func1(A, B);
 dmcall func2(A, result);
 emit B; // emitting B with a value stored in func2
}
```

```
// In ext_modules.c

void func2(signal *arg0, cfv *arg1) {
 /* TODO: implement this function */
}
```

SystemGALS – Program with C as a Host Language

```
#include “sysgals.h”
void sysgals_initialize() {
          sgl_ConnectChan(&__chan_CD1_C,&__chan_CD2_C);
}

int main (int argc, const char *argv[]){
  sysgals_initialize();
    while(1){
       CD1();
       CD2();
     }
}
```

More details in:

Salcic, Z., Park, H., Biglari-Abhari, M., and Teich, J., 2019, SystemGALS – A language for the design of GALS software systems, Embedded Systems Research Group, University of Auckland, Internal document, Embedded Systems Research Group, available to C701 class