COMPSYS701:Advanced Digital Design

Custom Computing Machines: Designing a Reactive Microprocessor

Zoran Salcic

Department of Electrical and Computer Engineering

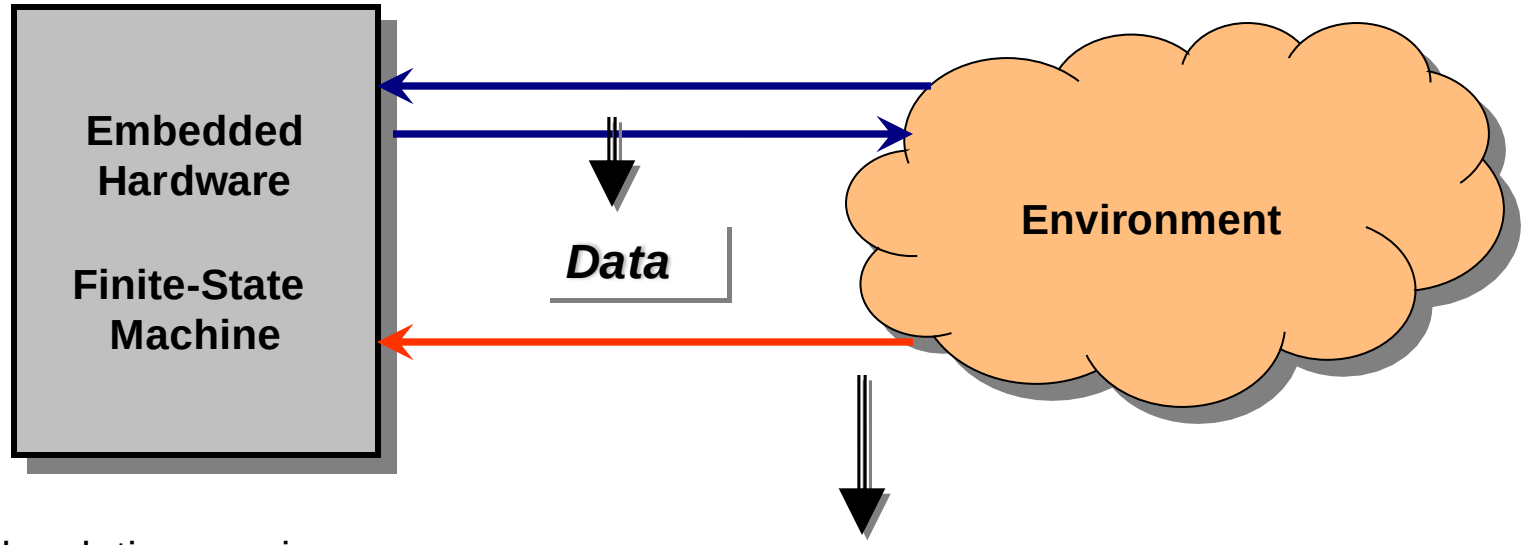
University of Auckland

Semester 1, 2025

In this presentation

- **We show the concepts and architecture of a custom-computing machine / customisable processor**
- **Approach to customisation**
- **Non-pipelined and pipelined non-reactive core (MiCORE)**
- Reactivity to external events with Reactive Functional Unit (RFU) (ReMIC)
- Reduced power/energy consumption Power Aware ReMIC (PAReMIC)
- Real-time operation with Scheduling Support Unit (SSU)
- Multiprocessing on a chip (HiDRA)

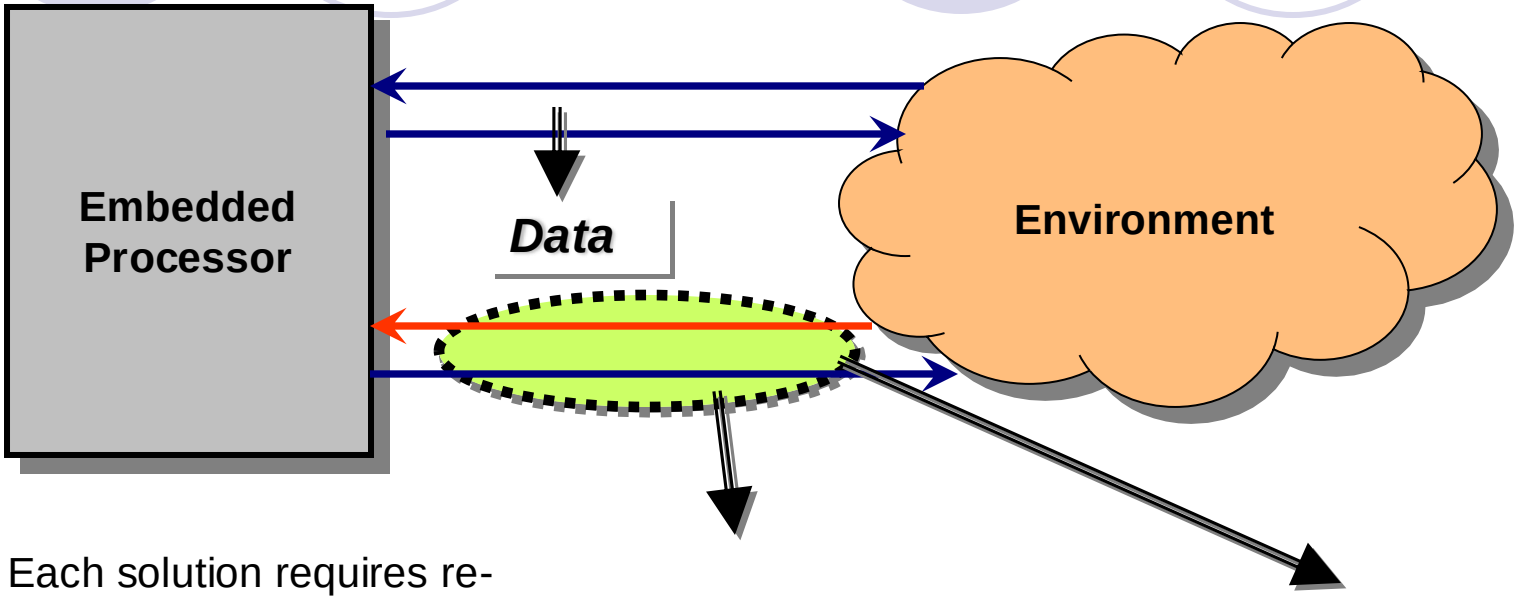
Traditional Dedicated Hardware-Based Reactive System



Each solution requires design of new hardware – it includes the full design cycle and implementation (ASIC or FPGA)

Checking for the presence of certain signals in the environment synchronously with the system clock – every clock cycle

Traditional Processor-based Reactive System



Each solution requires re-design of software only

Polling **mechanism**

Checking for the presence of certain signals in the environment routinely (periodically)

Interrupt **mechanism**

An alert mechanism when certain signals occur in the environment

Motivation for Reactive Processors

- A new set of features and native instructions for interaction with a external environment:
  - Avoid busy waiting associated with polling external signals/events, when required
  - Avoid context switching associated with interrupts
  - Enable efficient preemption mechanism
  - Provide predictable and guaranteed timing characteristics
  - Software controlled from the user point-of-view

Processor Reactivity Model

- Processors are inspired by ESTEREL model of reactivity by providing specific native instructions for delay, signal emission, priorities and preemption
- The same processor core can be used to implement different reactive algorithms for different applications by changing only programs not the processor hardware
- Preserves performance predictability by guaranteeing execution time for all primitive instructions

Esterel Example

```
abort
    loop
        await A;
        emit B;
        present C then
            emit D
        end;
        await tick; pause
    end
    ||
    loop
        present B then
            emit C
        end;
        await tick; pause
    end
when R
```

Preemption

Signal Polling

Conditional execution

Concurrent Execution

When A is present, B, C and D are also emitted to the environment at the same logical time known as a ‘tick’ in Esterel

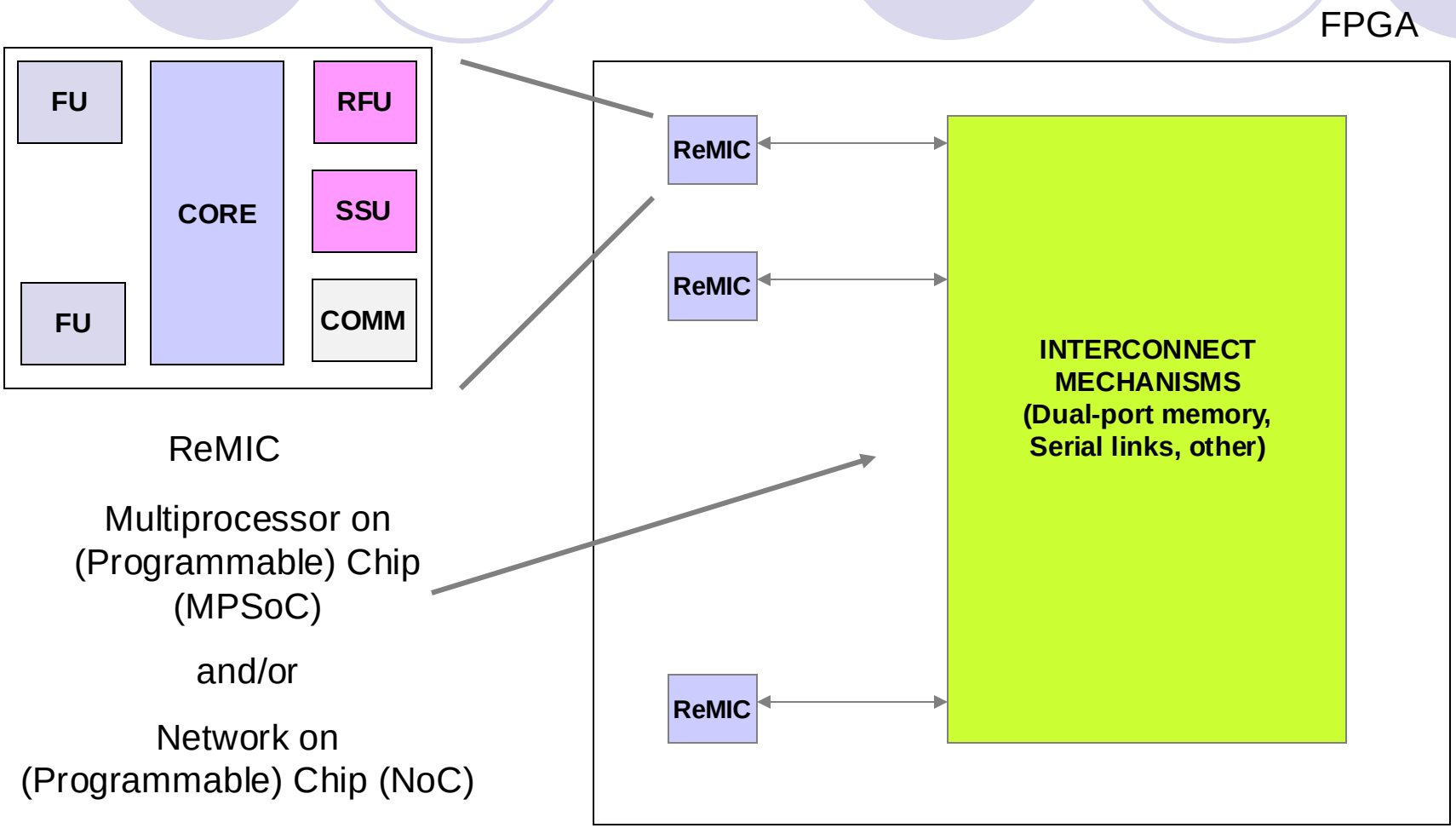
ReMIC – Reactive MICroprocesssor

- Efficient mapping of control-dominated applications on processor ISA - Better performance and code density for control-dominated (part of) applications.
- Support for direct execution of reactive language features – Support for language constructs implementation on machine level
- Support for concurrency and multiple processor configurations - Supports building multiple processors that execute multiple application processes and provide a mechanism for their synchronization and communication.

ReMIC

- Consists of:
  - Non-reactive processor core MiCORE + Reactive Functional Unit (FU) +
    - Scheduling Support Unit (SSU) for multiple processes
    - Specialised algorithms implemented in hardware functional units
  - Enables multi-core configurations

ReMIC – A New Processor Core Architecture for Heterogeneous Systems



ReMIC

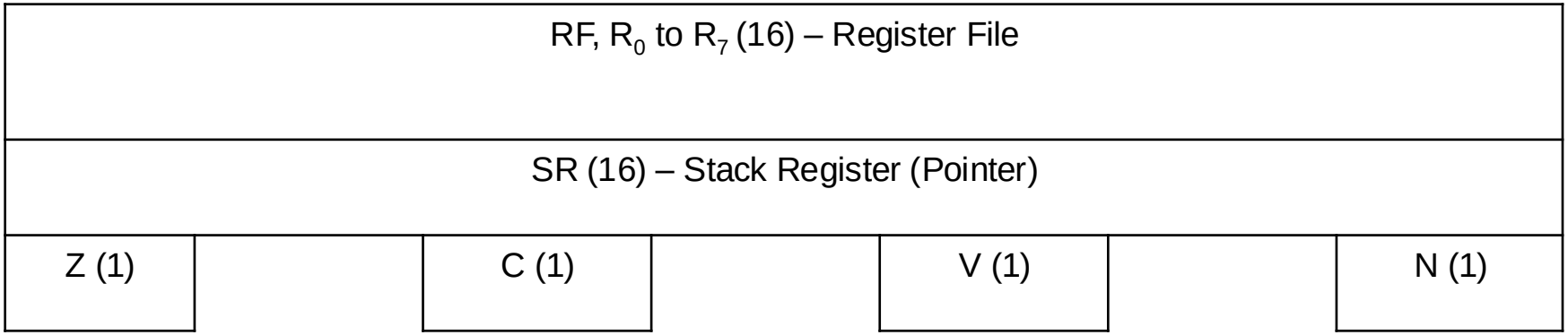
Multiprocessor on (Programmable) Chip (MPSoC)

and/or

Network on (Programmable) Chip (NoC)

MiCORE - Non-reactive Processor Core Programming Model

User-visible registers



Status Register Bits: Zero, Carry, Overflow, Negative

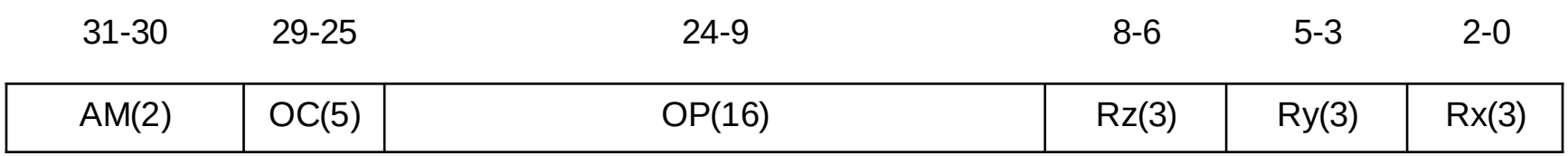
MiCORE Programming Model

- User-invisible registers: + POP, PPC, POC, PSX, PSY and PSZ pipeline registers (used in pipeline versions only)

<table>
<tr><td colspan="4">IR (32) – Instruction Register</td></tr>
<tr><td colspan="4">AR (16) – Address Register</td></tr>
<tr><td colspan="4">PC (16) – Program Counter</td></tr>
<tr><td colspan="4">POP (16) – Pipeline Operation Register</td></tr>
<tr><td colspan="4">PPC (16) – Pipeline Program Counter</td></tr>
<tr><td>POC (7) – P. OpCode</td><td>PSX (3) – P. RF Adr X</td><td>PSY (3) – P. RF Adr Y</td><td>PSZ (3) – P. RF Adr Z</td></tr>
</table>

MiCORE – Instruction Format

Instruction format



AM	=	Addressing Mode
OC	=	Opcode
OP	=	Operand
Rz	=	Destination register designator (R0 to R7)
Ry	=	Source register Y designator (R0 to R7)
Rx	=	Source register X designator (R0 to R7)

MiCORE – Addressing Modes

- Immediate (value a part of instruction)
- Inherent (or register) (operations on register content)
- Direct (Memory address a part of instruction)
- Register indirect (Memory address held in working registers)
- Stack (SR used for stack related operations)

- Instructions use up to three addresses/resources for the destination of the result and source of operands

MiCORE – Instruction Groups

- **Memory reference instructions** - use direct or register indirect addressing mode to transfer data between MiCORE and memory or I/O registers.
- **Register reference instructions** - operate on the contents of the MICORE registers or transfer the contents between them
- **Program flow control instructions** - enable conditional and unconditional change of program flow

MiCORE – Memory Reference Instructions

Memory Reference Instructions		
LDR Rz address	Direct	Rz ← M[address]
LDR Rz Ry	Indirect	Rz ← M[Ry]
STR Rx address	Direct	M[address] ← Rx
STR Rx Ry	Indirect	M[Ry] ← Rx
STR Ry #value	Indirect	M[Ry] ← value
LDSP address	Direct	SR ← M[address]
PSH Rx	Stack	M[SR] ← Rx, SR ← SR – 1
PUL Rz	Stack	SR ← SR + 1, Rz ← M[SR]

MiCORE – Register Reference Instructions

Register Reference Instructions		
LDR Rz #value	Immediate	Rz ← value
LDSP #value	Immediate	SR ← value
ADD Rz Ry Rx	Inherent	Rz ← Rx + Ry
ADD Rz Rx #value	Inherent	Rz ← Rx + value
SUB Rz Ry Rx	Inherent	Rz ← Rx – Ry
SUB Rz Rx #value	Inherent	Rz ← Rx – value
AND Rz Ry Rx	Inherent	Rz ← Rx AND Ry
AND Rz Rx #value	Inherent	Rz ← Rx AND value
OR Rz Ry Rx	Inherent	Rz ← Rx OR Ry
OR Rz Rx #value	Inherent	Rz ← Rx OR value
XOR Rz Ry Rx	Inherent	Rz ← Rx XOR Ry
XOR Rz Rx #value	Inherent	Rz ← Rx XOR value
CMP Rz Rx	Inherent	Rz ← NOT (Rx)

MiCORE – Register Reference Instructions

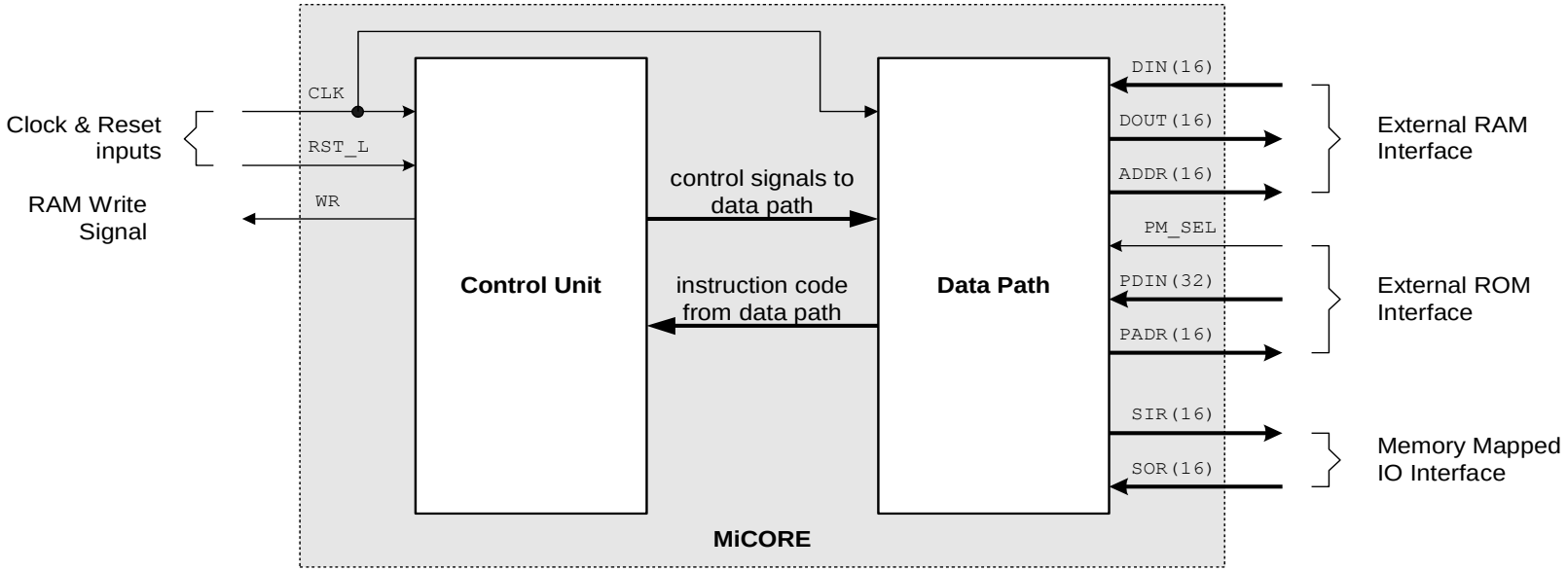
Register Reference Instructions		
LSH Rz Rx	Inherent	Rz[15..1] ← Rx[14..0], Rz[0] ← ‘0’
RSH Rz Rx	Inherent	Rz[14..0] ← Rx[15..1], Rz[15] ← ‘0’
LRT Rz Rx	Inherent	Rz[15..1] ← Rx[14..0], Rz[0] ← Rx[15]
RRT Rz Rx	Inherent	Rz[14..0] ← Rx[15..1], Rz[15] ← Rx[0]
ABS Rz Rx	Inherent	Rz ← |Rx|
TRF Rz Rx	Inherent	Rz ← Rx
CLF CZVN	Inherent	If C, Cflag ← 0 If Z, Zflag ← 0 If V, Vflag ← 0 If N, Nflag ← 0

MiCORE – Program Flow Control/Execution

Program Flow Control		
SC address	Immediate	If C = 1, PC ← address
SZ address	Immediate	If Z = 1, PC ← address
SV address	Immediate	If V = 1, PC ← address
SN address	Immediate	If N = 1, PC ← address
JMP address	Immediate	PC ← address
JMP Ry	Inherent	PC ← Ry
NOOP	Inherent	No operation
JSR address	Stack	M[SR] ← PC, SR ← SR – 1, PC ← address
RET	Stack	SR ← SR + 1, PC ← M[SR]

MiCORE – Top View and Interface

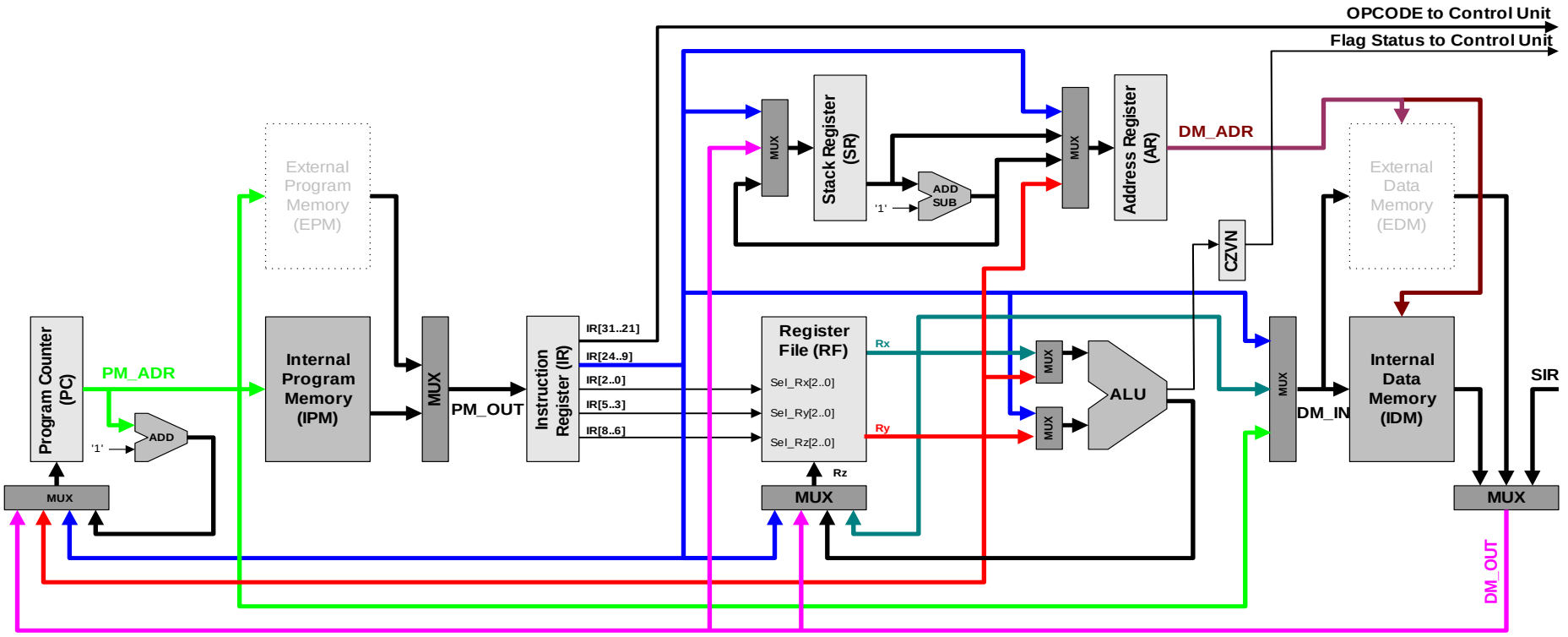
External view



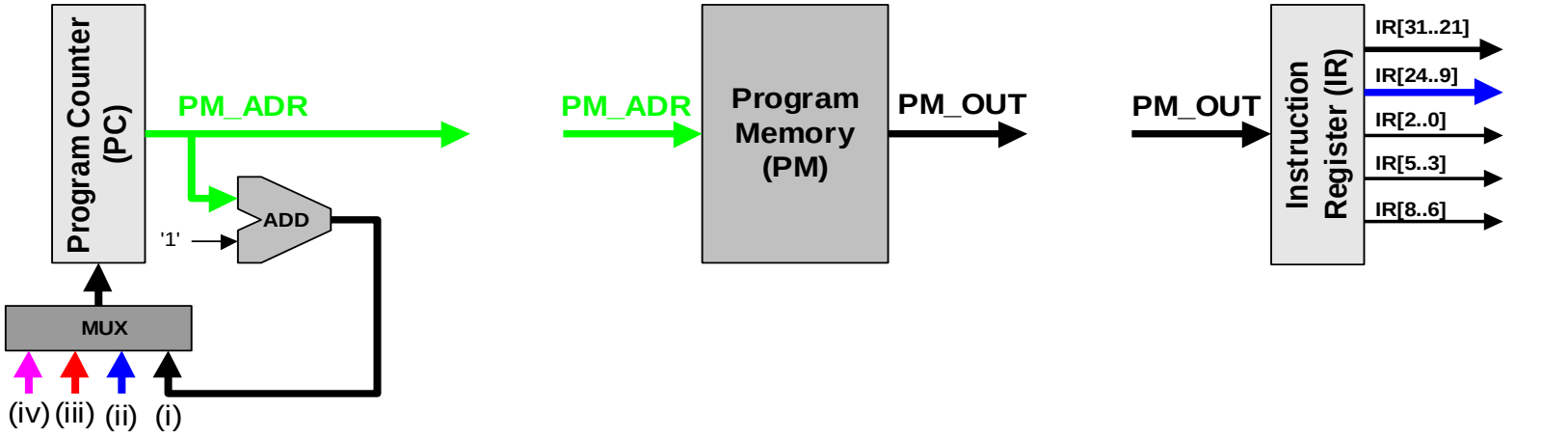
MiCORE - Datapath

- The data path contains eight functional blocks:
  - Program Counter (PC)
  - Program Memory (PM)
  - Instruction Register (IR)
  - Register File (RF)
  - Arithmetic Logic Unit (ALU)
  - Data Memory (DM)
  - Stack Register (SR)
  - Address Register (AR)

MiCORE – Datapath Organisation



MiCORE – Datapath Functional Blocks

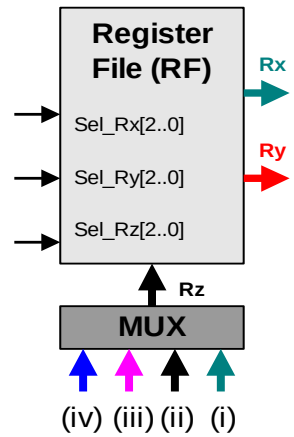


(a) Program Counter Block

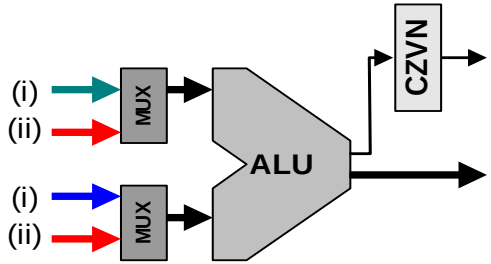
(b) Program Memory Block

(c) Instruction Register Block

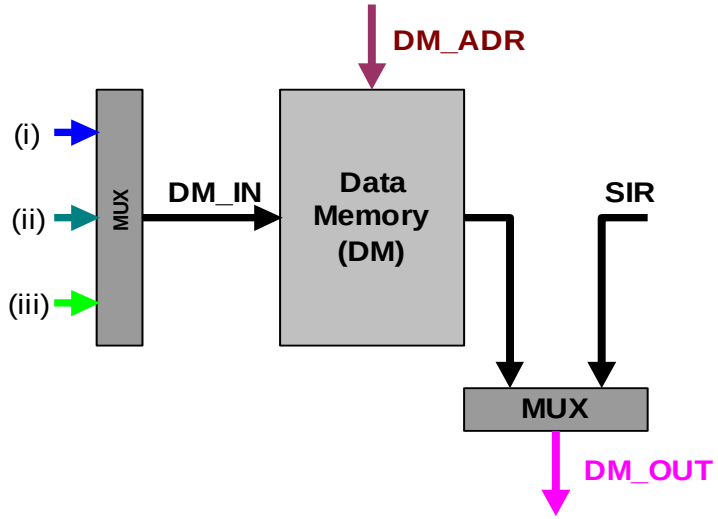
MiCORE – Datapath Functional Blocks



(a) Register File Block

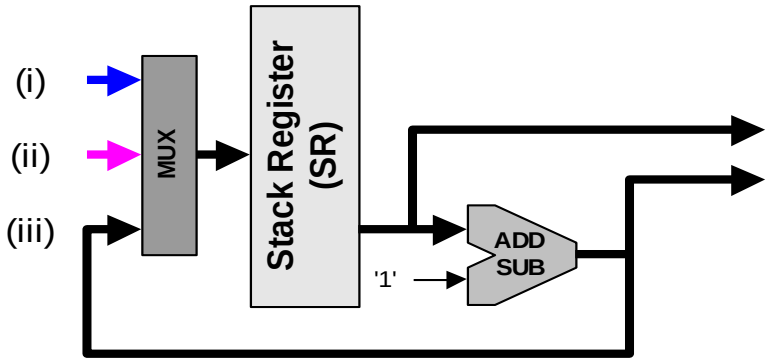


(b) Arithmetic Logic Unit Block

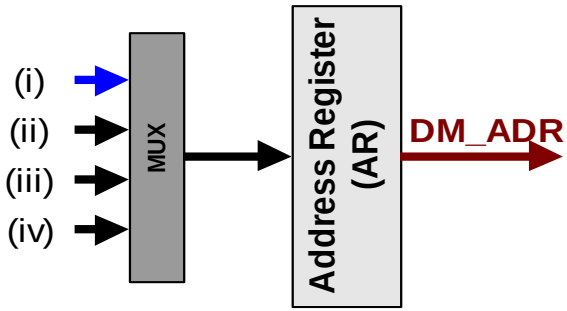


(c) Data Memory Block

MiCORE – Datapath Functional Blocks

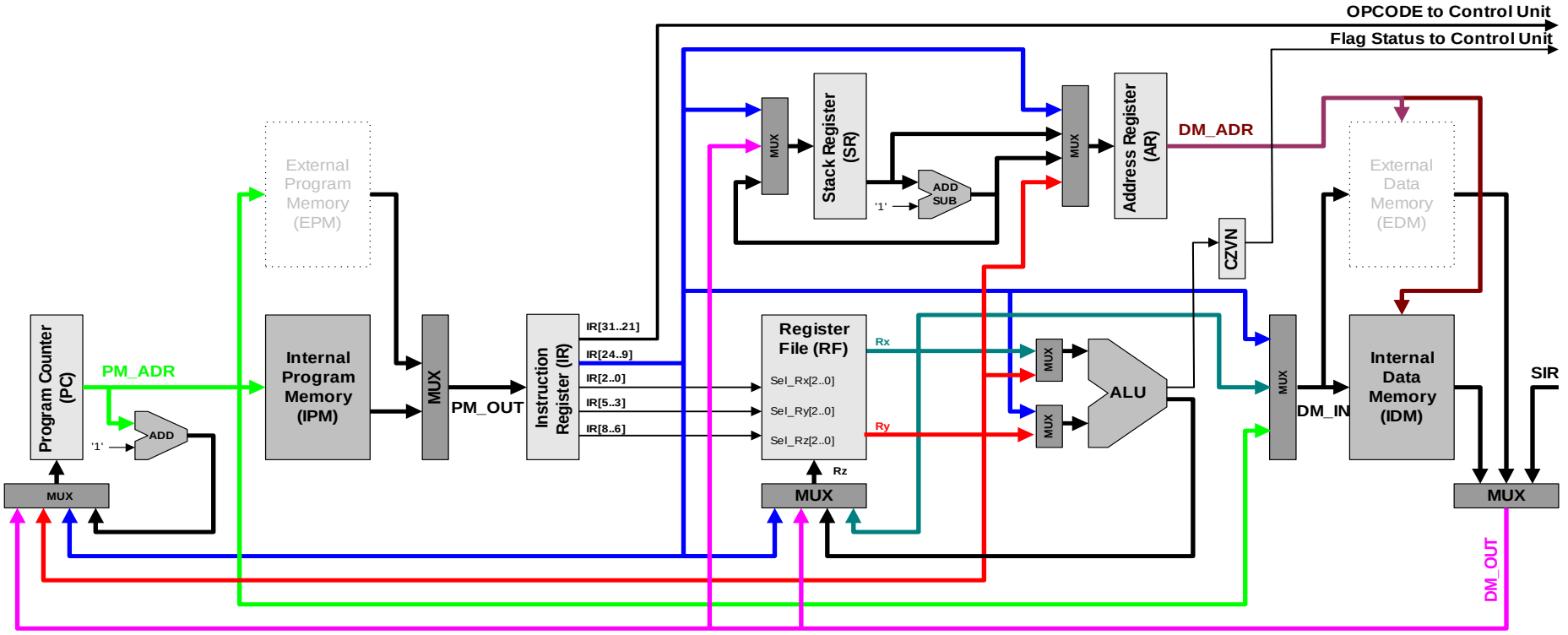


(a) Stack Pointer Block

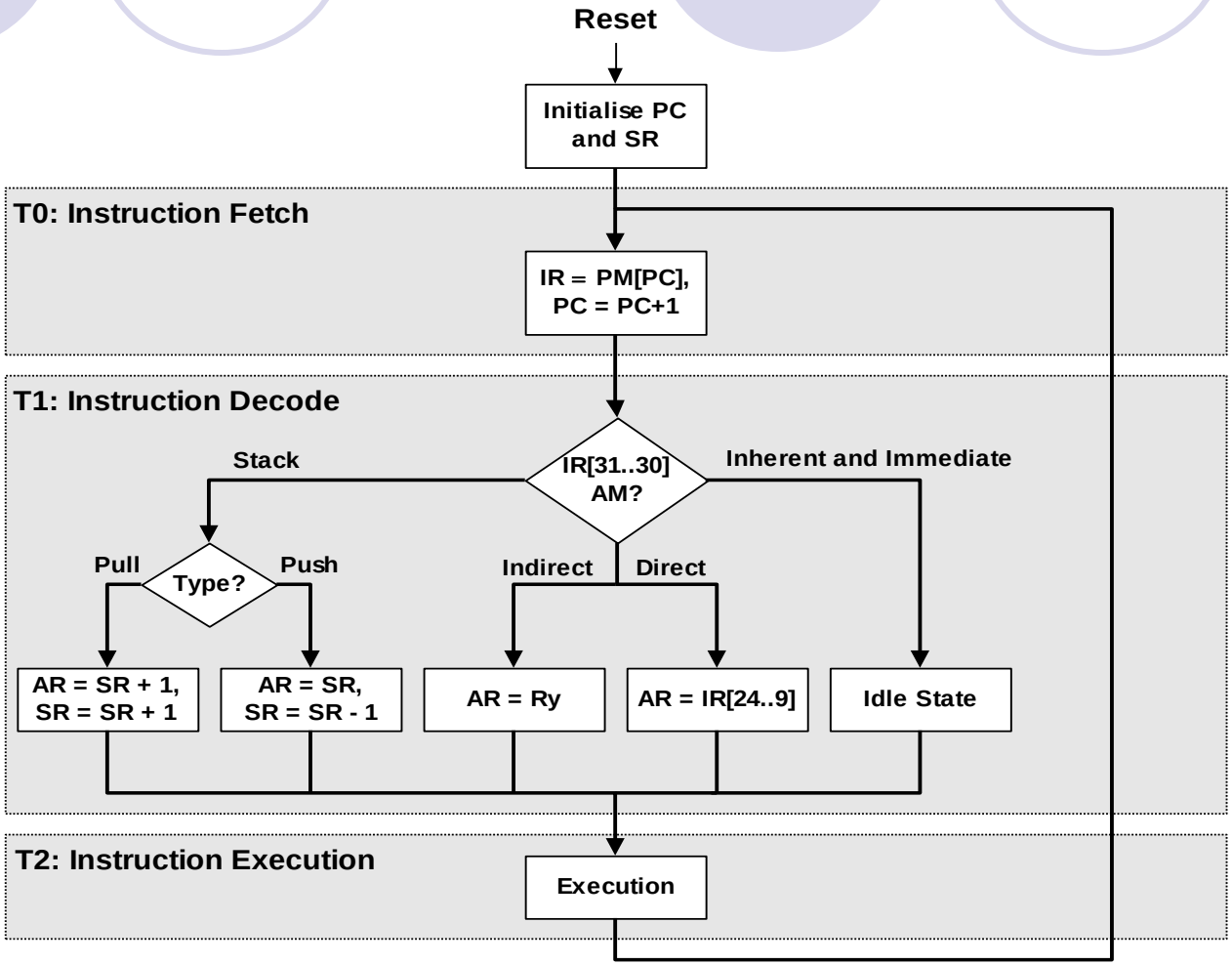


(b) Address Register Block

MiCORE – Integrated Datapath



MiCORE – Instruction Execution FSM



MiCORE – Instruction Execution Examples through Register Transfers

LDR Rz address	Direct	T1: AR ← IR[24..9] T2: Rz ← M[AR]
STR Rx Ry	Indirect	T1: AR ← Ry, T2: M[AR] ← Rx
STR Ry #value	Indirect	T1: AR ← Ry, T2: M[AR] ← IR[24..9]
LDSP address	Direct	T1: AR ← IR[24..9], T2: SR ← M[AR]
PSH Rx	Stack	T1: AR ← SR, T1: SR ← SR – 1, T2: M[AR] ← Rx

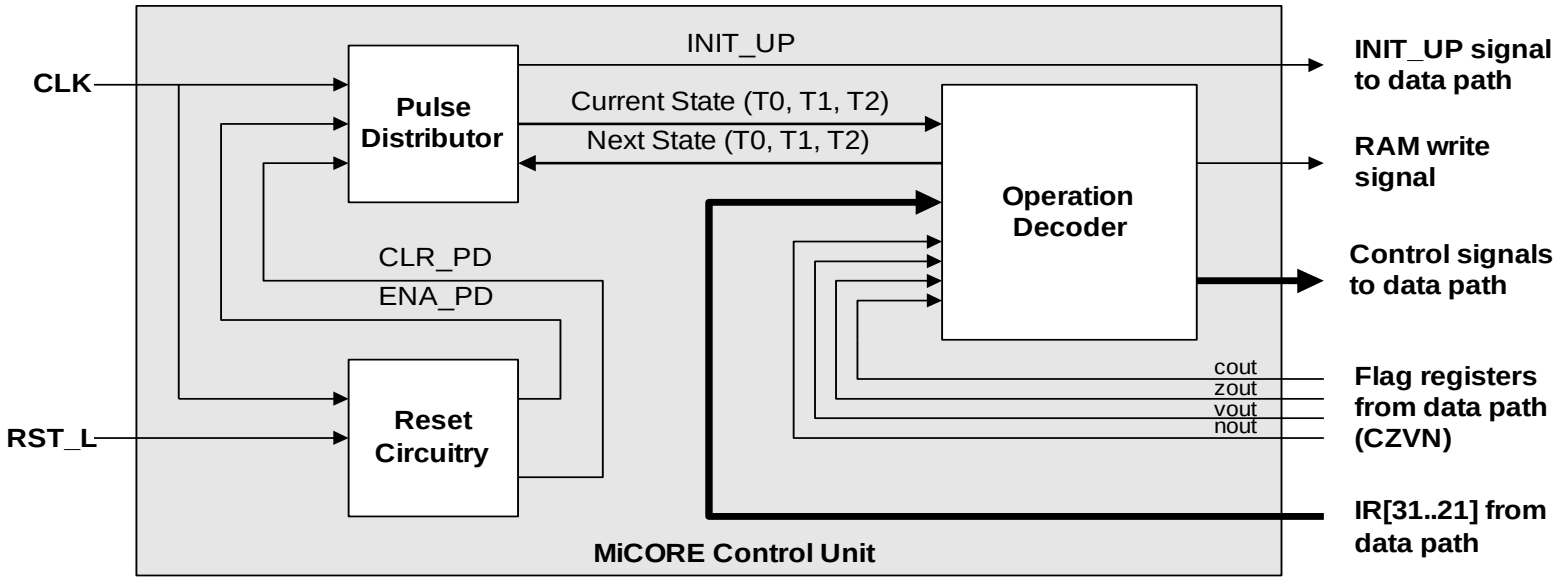
MiCORE – Instruction Execution Examples through Register Transfers

ADD Rz Ry Rx	Inherent	T1: No Operation T2: Rz ← Rx + Ry
SUB Rz Rx #val	Inherent	T1: No Operation T2: Rz ← Rx – IR[24..9]
AND Rz Ry Rx	Inherent	T1: No Operation T2: Rz ← Rx AND Ry
OR Rz Ry Rx	Inherent	T1: No Operation T2: Rz ← Rx OR Ry

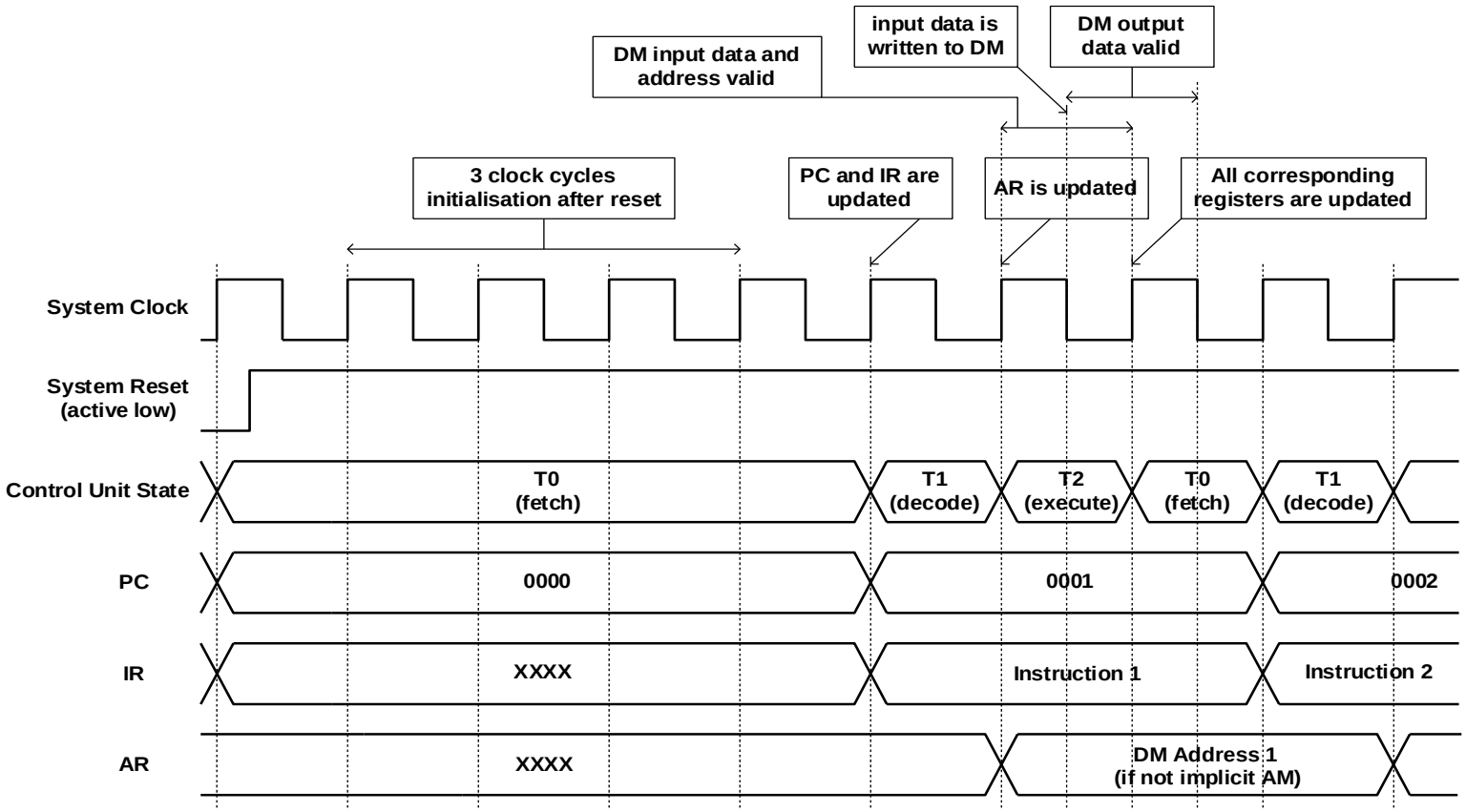
MiCORE – Instruction Execution Examples through Register Transfers

SC address	Inherent	T1: No Operation T2: If C = 1, PC ← IR[24..9]
SZ address	Inherent	T1: No Operation T2: If Z = 1, PC ← IR[24..9]
JMP address	Immediate	T1: No Operation T2: PC ← IR[24..9]
JMP Ry	Inherent	T1: No Operation T2: PC ← Ry
NOOP	Inherent	T1: No operation T2: No Operation
JSR address	Stack	T1: AR ← SR, T1: SR ← SR – 1, T2: M[AR] ← PC, T2: PC ← IR[24..9]
RET	Stack	T1: AR ← SR + 1, T1: SR ← SR + 1, T2: PC ← M[AR]

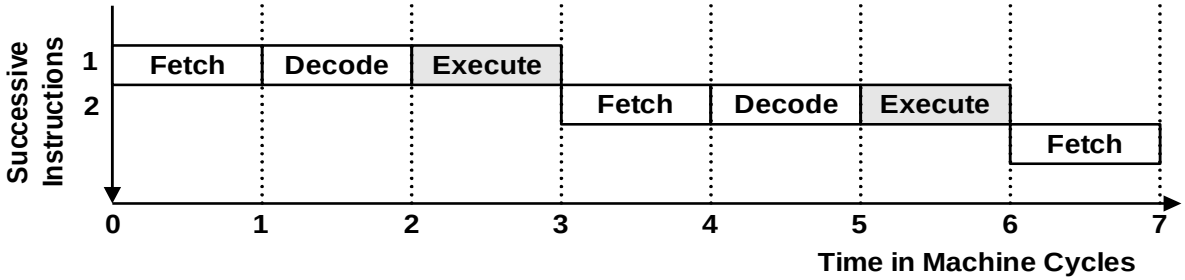
MiCORE – Control Unit Top View

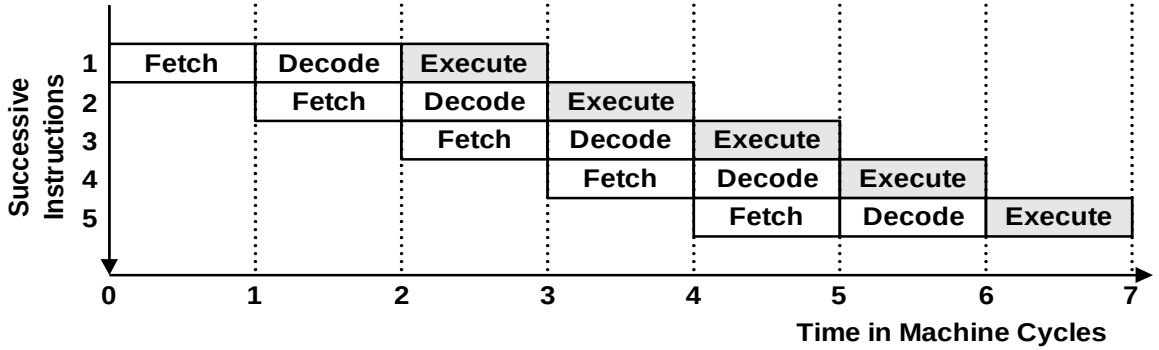


MiCORE – Timing Diagram Example

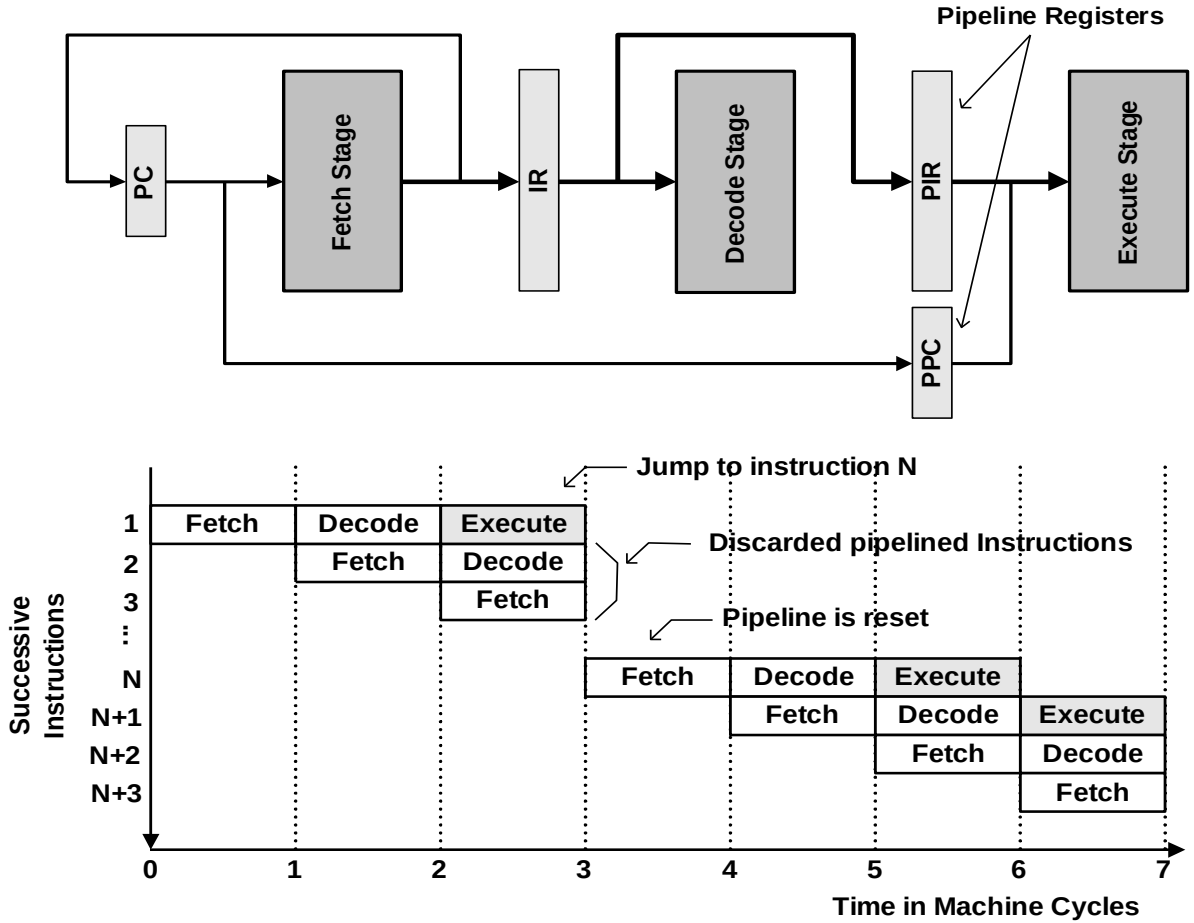


MiCORE - Non-pipelined vs pipelined execution



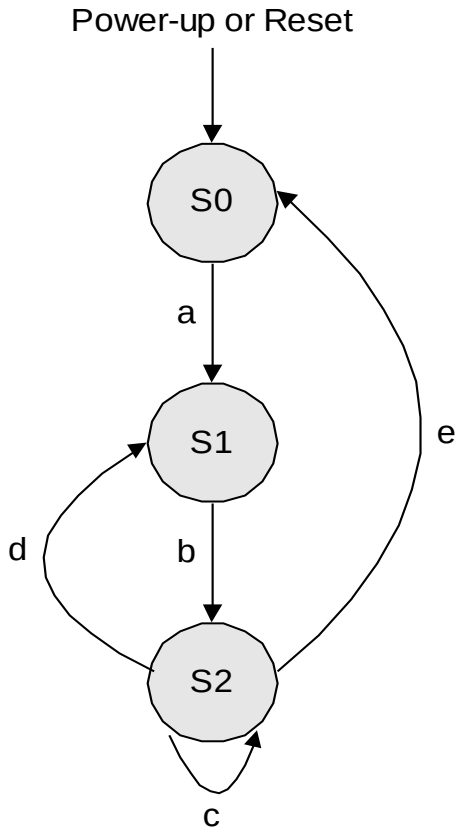


MiCORE - Pipelined core design



MiCORE - Pipelined core design

Pipelined Control Unit



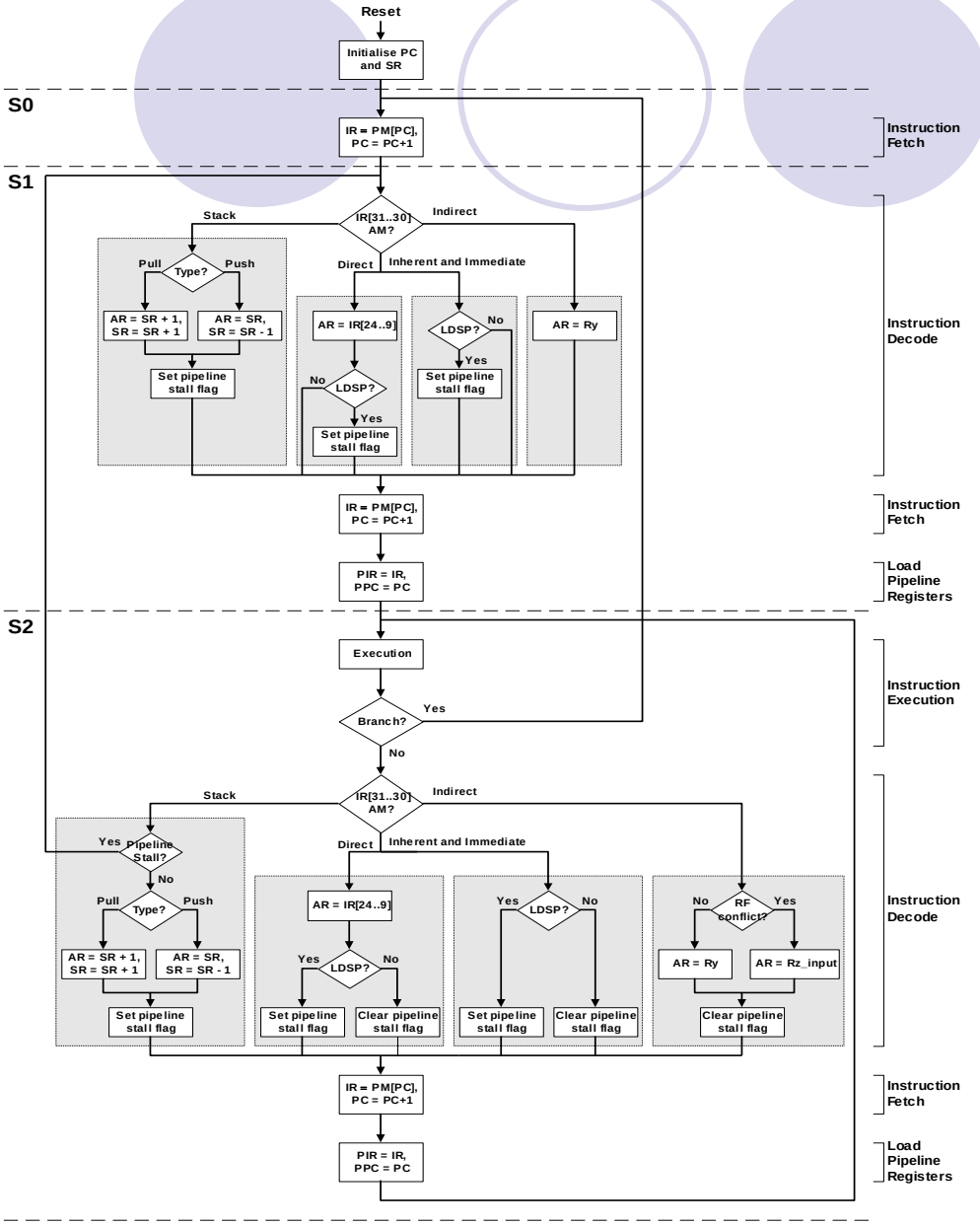
Operations performed at:
S0
(1) Instruction Fetch

S1
(1) Load Pipeline Registers
(2) Instruction Decode
(3) Instruction Fetch

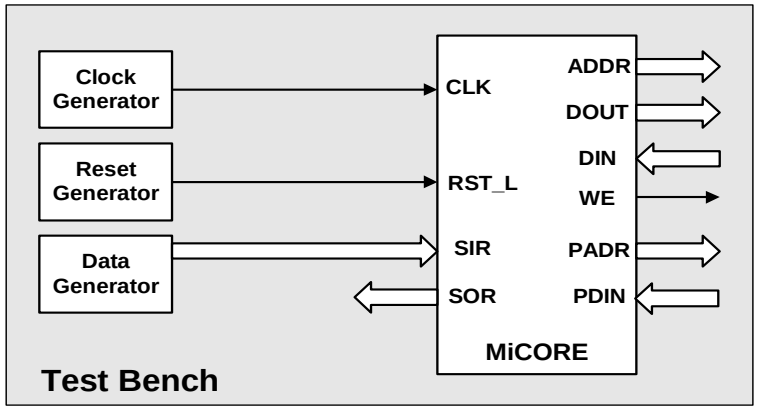
S2
(1) Load Pipeline register
(2) Instruction Execution
(3) Instruction Decode
(4) Instruction Fetch

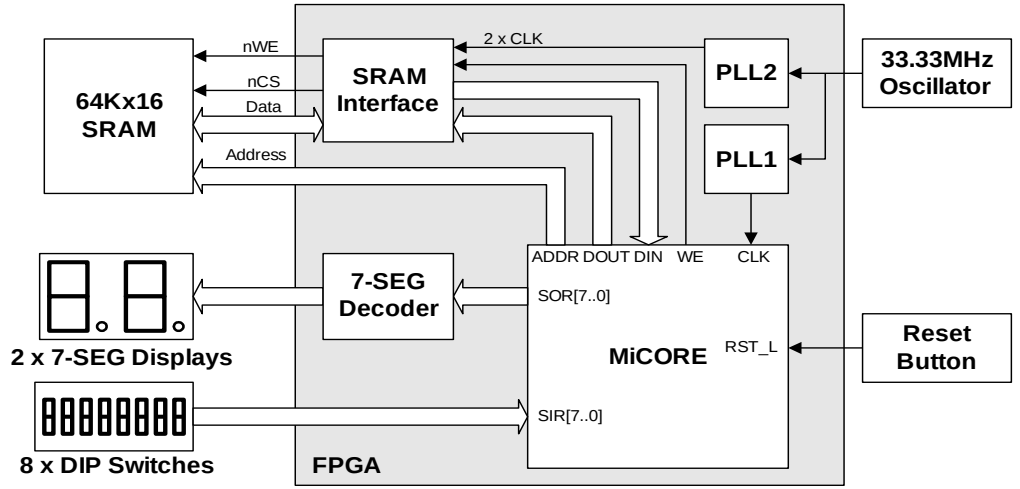
State Transition Condition:
(a) Rising edge of the system clock
(b) Rising edge of the system clock
(c) Rising edge of the system clock
(d) When a pipeline stall condition is detected
(e) When a branching instruction is executed

MiCORE – Pipelined Control Unit FSM

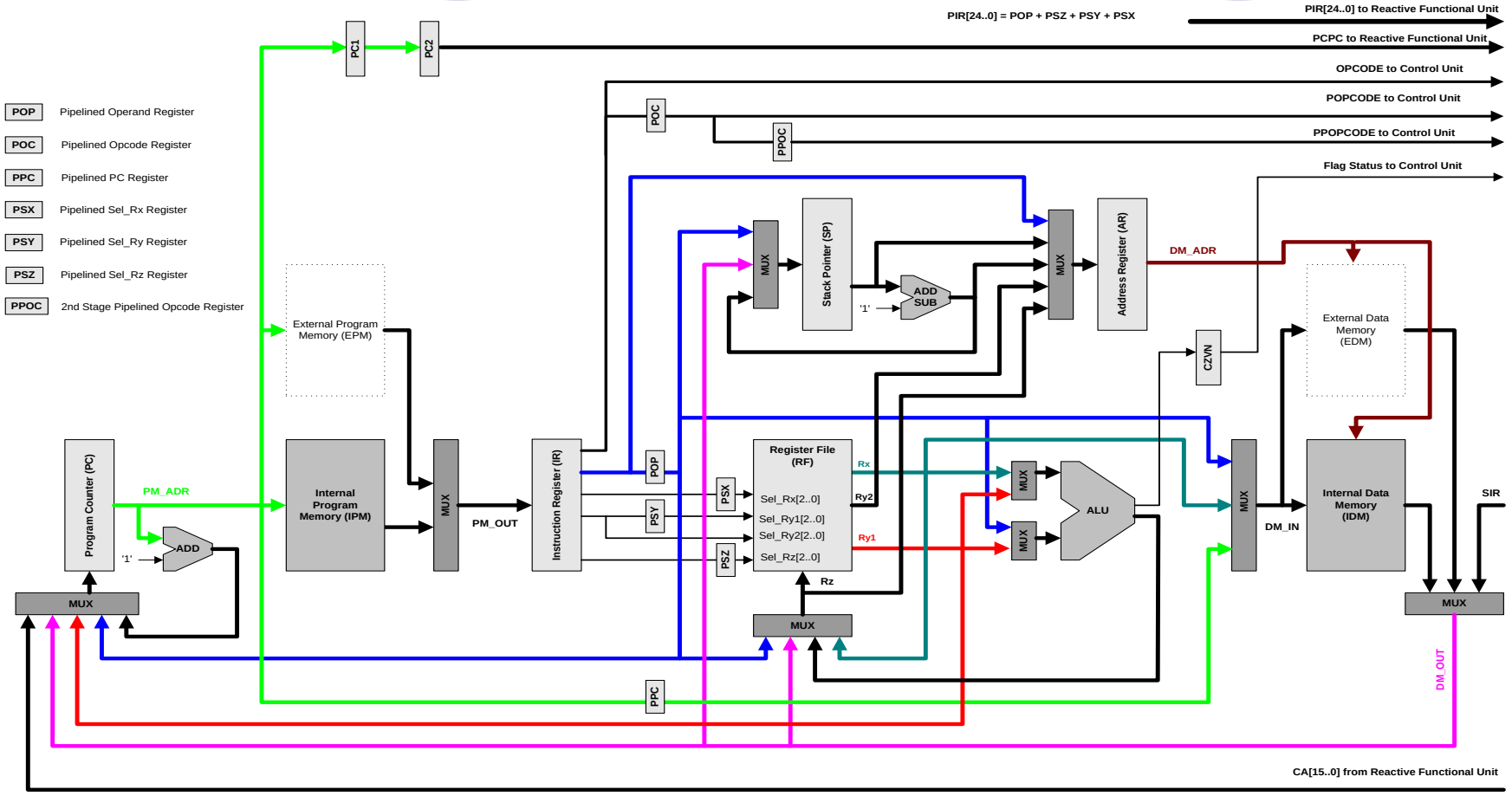


MiCORE -Testbench and FPGA Implementation





Pipelined ReMIC – Datapath



ReMIC – Reactive MICroprocessor

ReMIC – Instructions Supporting Reactivity

Features	Instruction Syntax
Signal emission	**EMIT** *signal(s)*
Signal sustenance	**SUSTAIN** *signal(s)*
Signal polling	**SAWAIT** *signal*
Delay	**TAWAIT** *clock(s)*, *prescaler*
Conditional signal polling	**CAWAIT** *signal1*, *signal2*, *address*
Signal presence	**PRESENT** *signal*, *address*
Preemption	**ABORT** *signal*, *address*

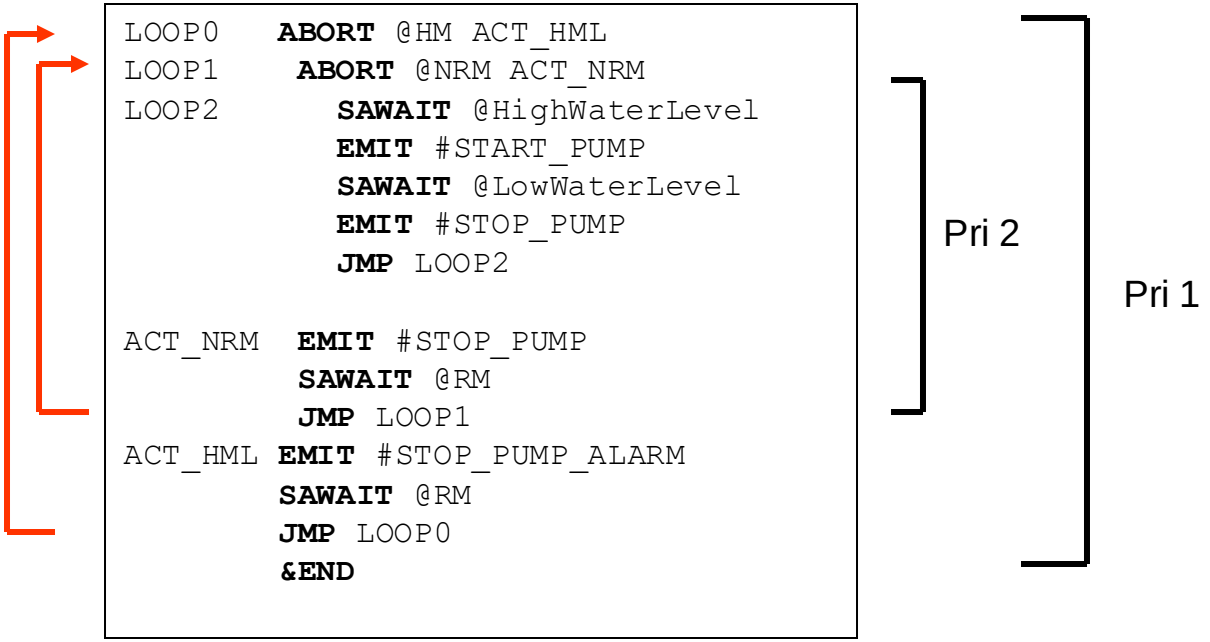
ReMIC – Example Program

"A pump controller … The pump is used to pump out water (whenever the water level exceeds the desired level) provided the methane level is below the desired level (RIGHT-METHANE RM).

Whenever, methane level goes above this desired level (NOT-RIGHT-METHANE NRM), the controller must stop the pump and wait until right methane level is restored.

If at any time, however, the methane level is too high (HIGH -METHANE HM) then the pump must be stopped immediately and an ALARM must be generated. Pumping is stopped until right methane level is restored".

```
HM&EQU 0 ; SIP[0]
NRM&EQU 1 ; SIP[1]
HighWaterLevel&EQU 2 ; SIP[2]
LowWaterLevel&EQU 3 ; SIP[3]
RM&EQU 4 ; SIP[4]
START_PUMP&EQU $0001 ; SOP[0]
STOP_PUMP&EQU $0002 ; SOP[1]
STOP_PUMP_ALARM&EQU $0006 ; SOP[2]=ALARM
```



ReMIC – Quantitative Results

Non-reactive vs Reactive Processor Application Memory Footprint (16-bit words)

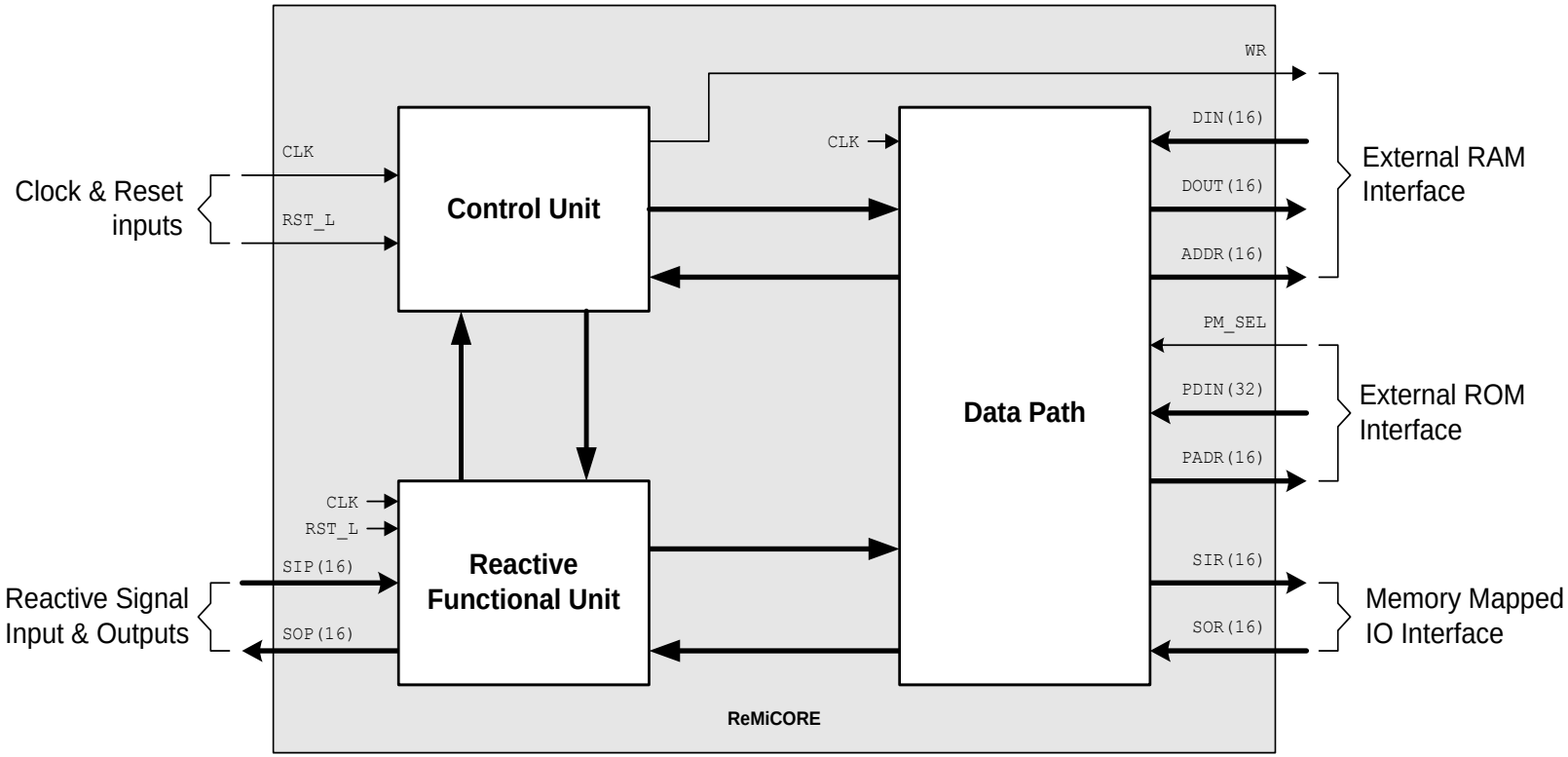
Application	**Abort Levels**	**Non-reactive core**	**ReMIC**
Seat Belt Controller	1	23	6
Pump Controller	2	60	13
Elevator Controller	0	86	25
Traffic Light Controller	2	68	16

ReMIC – Quantitative Results

Memory Footprint Comparison with Traditional Microprocessors (bytes)

Application	ReMIC	8051	68HC11	NIOS16
Pump Controller	26	35	56	80
Elevator Controller	50	45	79	116
Traffic Light Controller	32	70	114	147

ReMIC Core


WR
CLK
RST_L
Clock & Reset inputs
Control Unit
CLK
Data Path
DIN(16)
DOUT(16)
ADDR(16)
External RAM Interface
PM_SEL
PDIN(32)
PADR(16)
External ROM Interface
CLK
RST_L
SIP(16)
SOP(16)
Reactive Signal Input & Outputs
Reactive Functional Unit
SIR(16)
SOR(16)
Memory Mapped IO Interface
ReMiCORE

ReMIC – Instructions Supporting Reactivity

Features	Instruction Syntax
Signal emission	**EMIT** *signal(s)*
Signal sustenance	**SUSTAIN** *signal(s)*
Signal polling	**SAWAIT** *signal*
Delay	**TAWAIT** *clock(s)*, *prescaler*
Conditional signal polling	**CAWAIT** *signal1*, *signal2*, *address*
Signal presence	**PRESENT** *signal*, *address*
Preemption	**ABORT** *signal*, *address*

ReMIC – Instructions Supporting Reactivity

Signal emission **EMIT** *signal(s)*

31-30	29-23	24-9	8-0
AM(2)	OC(5)	Signals(16)	Unused(4)

Signal sustenance **SUSTAIN** *signal(s)*

31-30	29-23	24-9	8-0
AM(2)	OC(5)	Signals(16)	Unused(4)

ReMIC – Instruction Supporting Reactivity

Signal polling **SAWAIT** *signal*

31-30	29-23	24-9	8-5	4-0
AM(2)	OC(5)	Unused(16)	SIG(4)	Unused(4)

Signal delay **TAWAIT** *clock(s)*, *prescaler*

31-30	29-23	24-9	8-5	4-0
AM(2)	OC(5)	Timer Clock Cycles(16)	PS(4)	Unused(4)

ReMIC – Instruction Supporting Reactivity

Conditional signal polling **CAWAIT** *signal1*, *signal2*, *address*

31-30	29-23	24-9	8-5	4-1	0
AM(2)	OC(5)	SIG2 Continuation Address(16)	SIG1(4)	SIG2(4)	

Signal presence **PRESENT** *signal*, *address*

31-30	29-23	24-9	8-5	4-0
AM(2)	OC(5)	Continuation Address(16)	SIG(4)	Unused(5)

ReMIC – Instruction Supporting Reactivity

Preemption **ABORT** *signal*, *address*

31-30	29-23	24-9	8-5	4-0
AM(2)	OC(5)	Continuation Address(16)	SIG(4)	Unused(5)

Timer Activation TSTART (similar to TAWAIT, but program execution continues)

31-30	29-23	24-9	8-5	4-0
AM(2)	OC(5)	Timer Clock Cycles(16)	PS(4)	Unused(5)

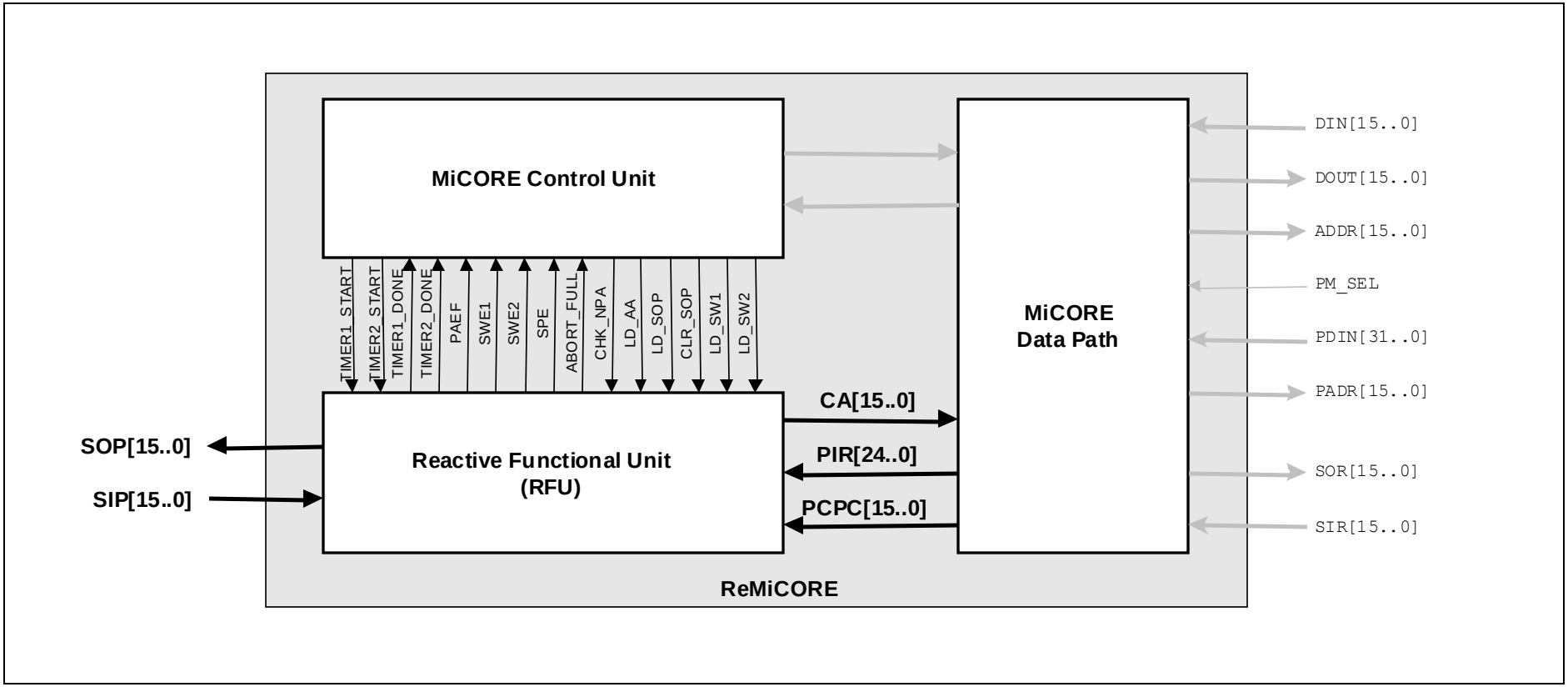
ReMIC – Reactive Instruction Execution

Instruction Mnemonic	**Address Mode**	**Instruction Execution in S2**
EMIT #signals	Functional Unit	SOP ← IR[24..9]
SUSTAIN #signals	Functional Unit	SOP ← IR[24..9]
SAWAIT S*i*	Functional Unit	SWR1 ← IR[8..5]
CAWAIT S*i* S*j* address	Functional Unit	SWR1 ← IR[8..5] SWR2 ← IR[4..1] SWA2 ← IR[24..9]
TAWAIT #cycles PS*i*	Functional Unit	TIMER_REF ← IR[24..9] CLK_SEL ← IR[8..5]
PRESENT S*i* address	Functional Unit	If S*i* is not present, PC ← IR[24..9]
ABORT S*i* address	Functional Unit	AAAR ← IR[24..9] AASR ← IR[8..5]
*S*i* or S*j* means Signal *i* or *j*, where *i* or *j* is an integer between 0 and 15. PS*i* means Prescaler Setting *i*, where *i* is an integer between 0 and 7.		

ReMIC – Reactive Program Example

```
BEGIN    ABORT S0 RESET              ;load first ABORT (1st nested) with the abort signal SIP[0]
           SAWAIT S1                 ;wait for the occurrence of SIP[1]
           EMIT #$1                  ;when SIP[1] occurs, Emit SOP[0]
           ABORT S2 CONT2            ;load second ABORT (2nd nested) with the abort signal SIP[2]
             CAWAIT S3 S4 CONT1      ;if SIP[3] occurs, continue, if SIP[4] occurs, go to CONT1,
                                     ; else wait
             LDR R1 #$1111           ;load R1 with a value $1111
CONT1        LDR R2 #$2222           ;load R2 with a value $2222
             PRESENT S4 CONT2        ;if SIP[4] is present, continue, else go to CONT2
             LDR R3 #$3333           ;load R3 with a value $3333
CONT2      LDR R4 #$4444             ;load R4 with a value $4444 and terminate the second ABORT
           ABORT S2 CONT4            ;load third ABORT (2nd nested) with the abort signal SIP[2]
             ABORT S3 CONT3          ;load fourth ABORT (3rd nested) with the abort signal SIP[3]
               TAWAIT0 #3 PS0        ;start timer 0 with a value $3 at the system clock divided by 2
               SUSTAIN #$2           ;sustain SOP[1] and halt the program execution
CONT3          LDR R6 #$6666         ;when SIP[3] occurs (fourth ABORT), load R6 with a value $6666
               SUSTAIN #$4           ;sustain SOP[2] and halt the program execution
CONT4      LDR R5 #$5555             ;when SIP[4] occurs (third ABORT), load R5 with a value $5555
RESET    JMP BEGIN                   ;when SIP[0] occurs (first ABORT) or the program executes to
                                     ; this point, jump to BEGIN
         &END
```

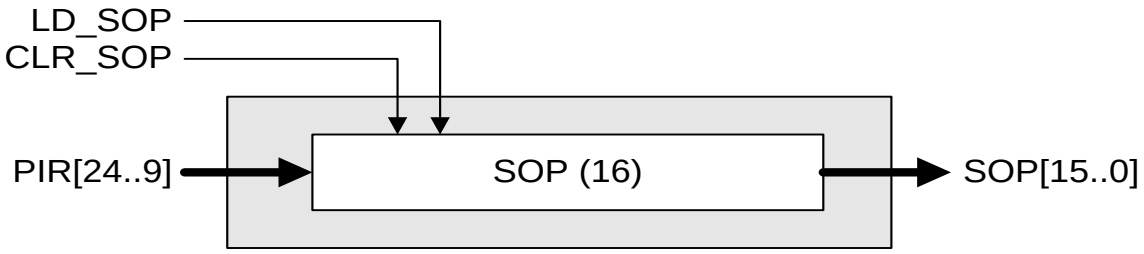
ReMIC – Top View



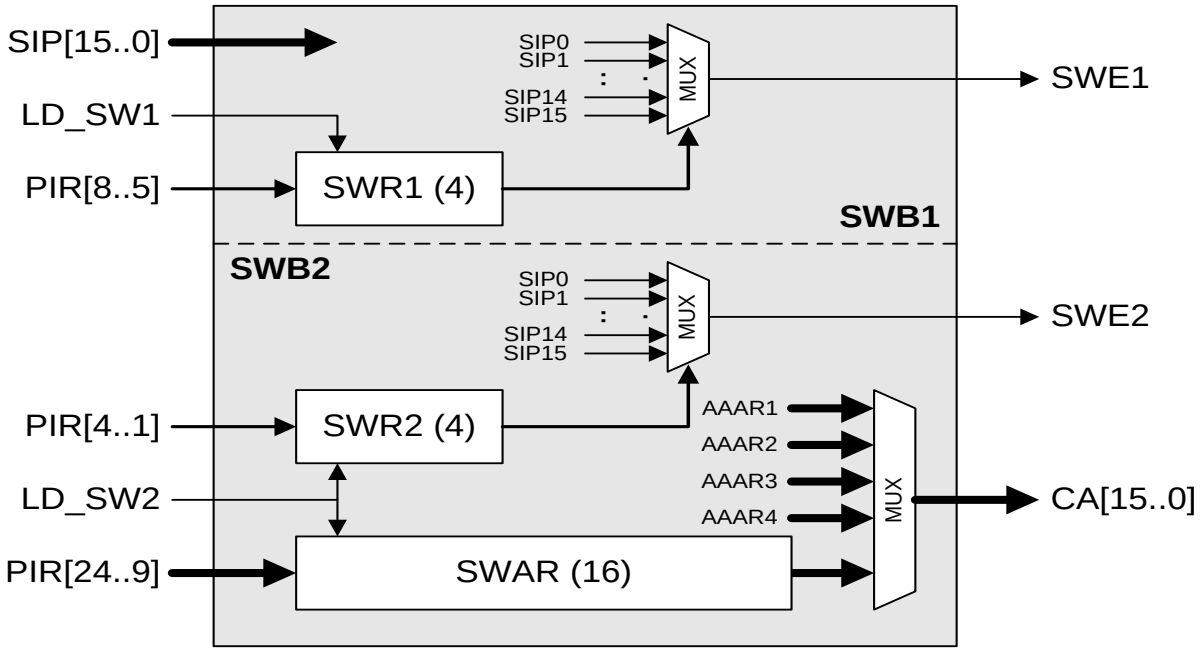
ReMIC Reactive Functional Unit

- The RFU is built up with the following five functional blocks:
  - Signal emission block
  - Signal wait block
  - Signal presence block
  - Timer block
  - Abort handling block

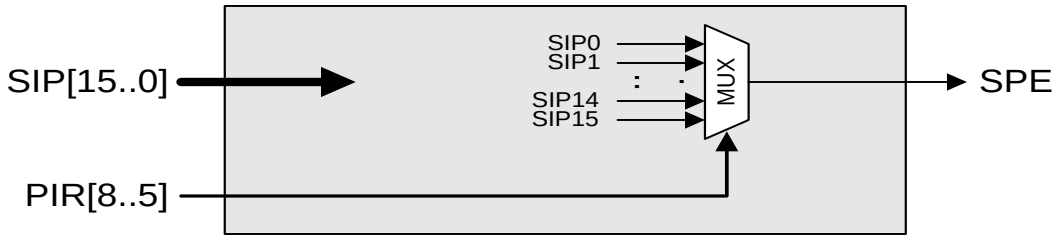
ReMIC RFU – Signal Emission Block



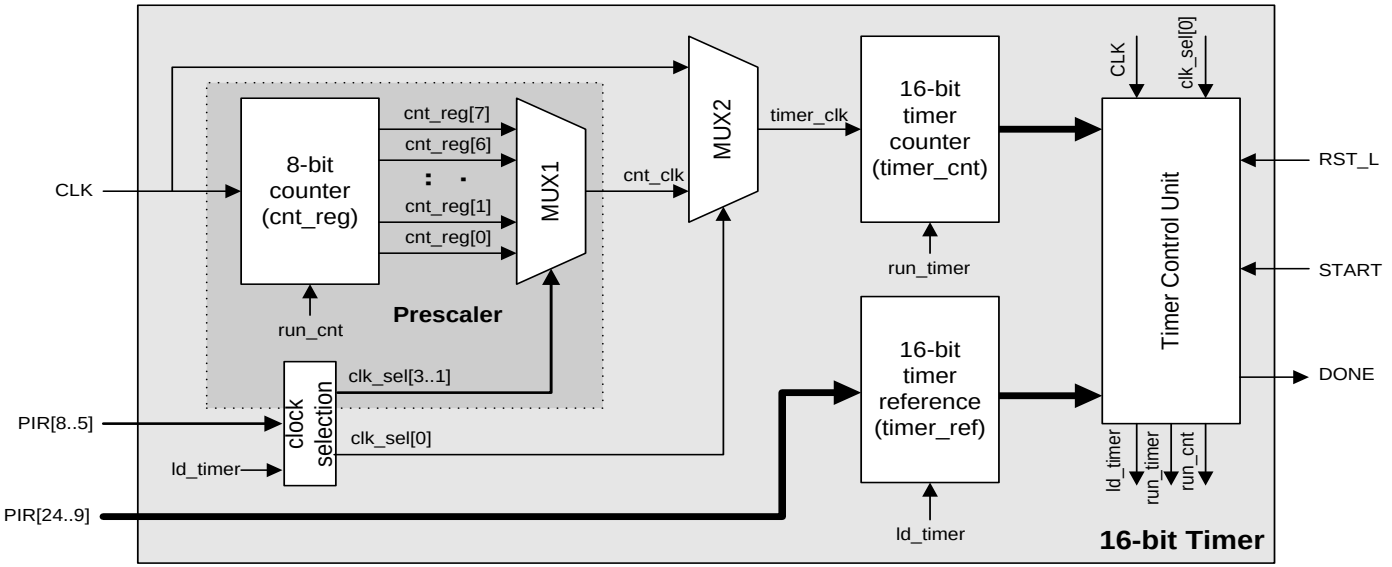
ReMIC RFU – Signal Wait Block

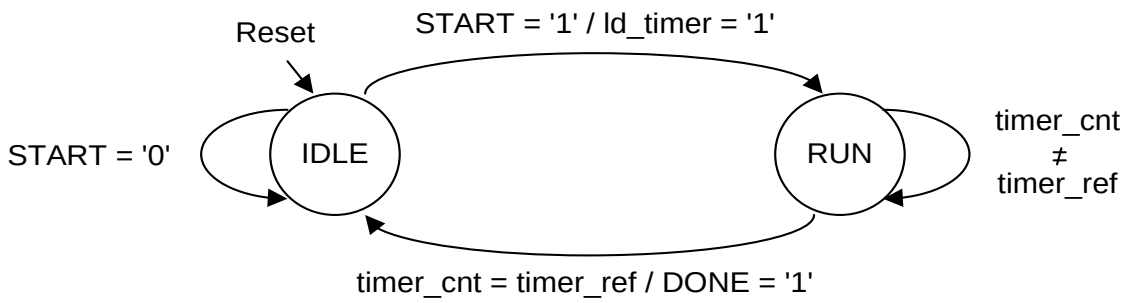


ReMIC RFU – Signal Presence Block

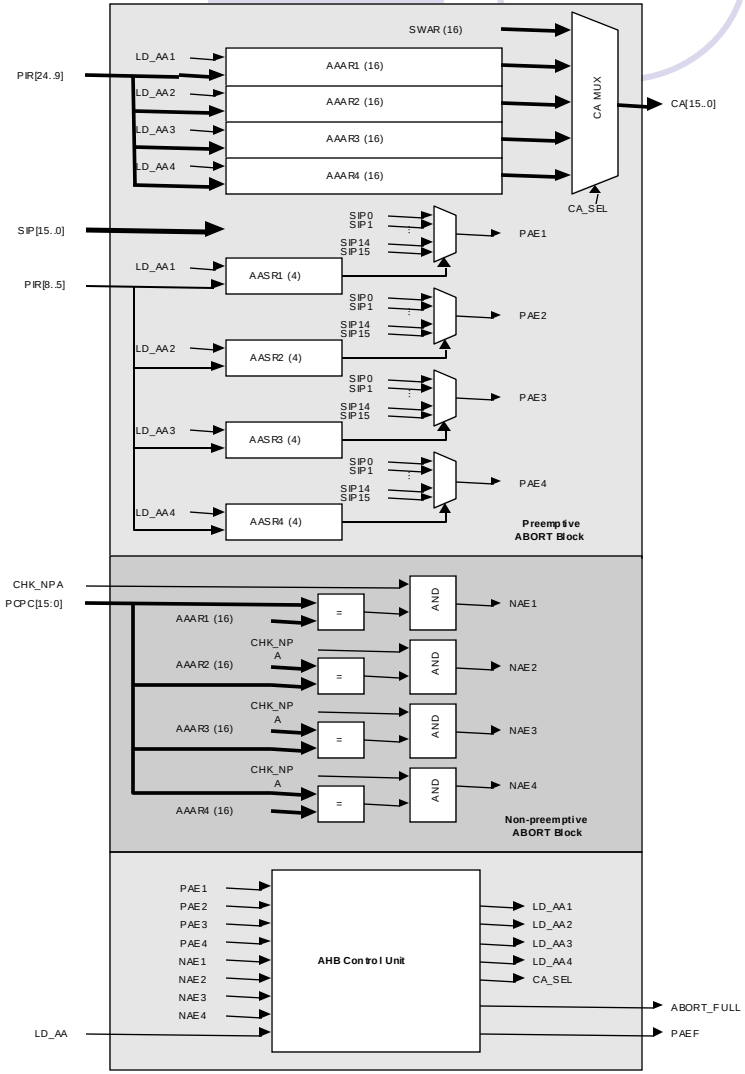


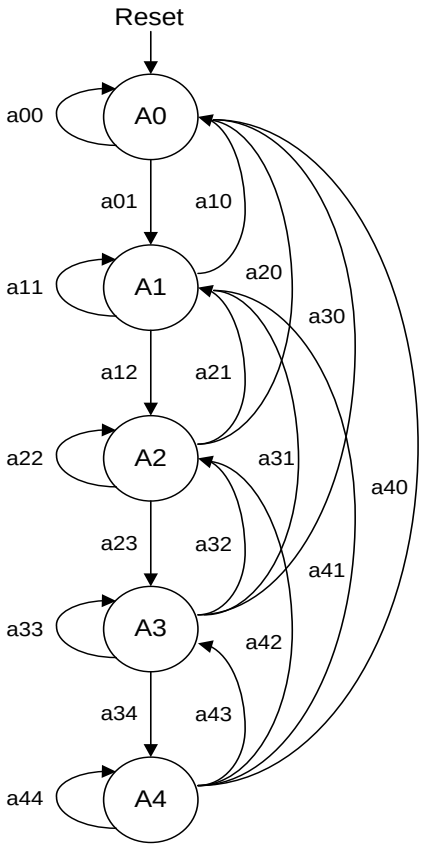
ReMIC RFU – Timer Block





ReMIC – Abort Handling





State Description:

A0: Abort empty (no ABORT loaded)
A1: Abort Level 1 (AASR1 and AAAR1 loaded with 1st ABORT)
A2: Abort Level 2 (AASR2 and AAAR2 loaded with 2nd ABORT)
A3: Abort Level 3 (AASR3 and AAAR3 loaded with 3rd ABORT)
A4: Abort Level 4 (AASR4 and AAAR4 loaded with 4th ABORT)

State Transition Condition:

a00: LD_AA = '0'
a01: LD_AA = '1'

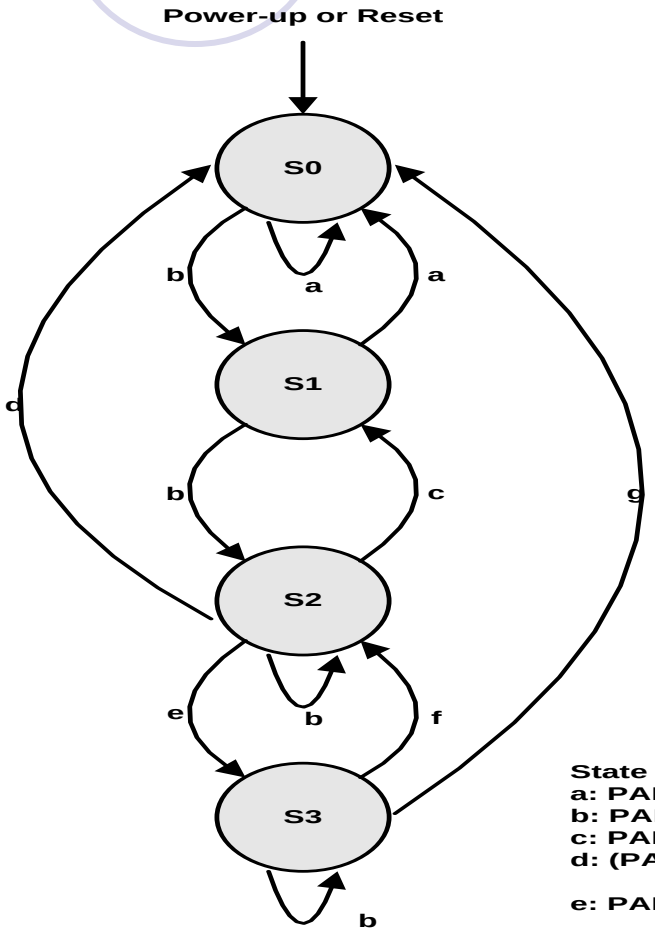
a10: NAE1 = '1' or PAE1 = '1'
a11: No operation
a12: LD_AA = '1'

a20: NAE1 = '1' or PAE1 = '1'
a21: NAE2 = '1' or PAE2 = '1'
a22: No operation
a23: LD_AA = '1'

a30: NAE1 = '1' or PAE1 = '1'
a31: NAE2 = '1' or PAE2 = '1'
a32: NAE3 = '1' or PAE3 = '1'
a33: No operation
a34: LD_AA = '1'

a40: NAE1 = '1' or PAE1 = '1'
a41: NAE2 = '1' or PAE2 = '1'
a42: NAE3 = '1' or PAE3 ='1'
a43: NAE4 = '1' or PAE4 = '1'
a44: No operation

ReMIC – Control Unit State Transitions



Operations performed at:

S0
if PAEF = '0' then
  (1) Instruction Fetch
else
  (1) load PC with abort
      continuation address

S1
if PAEF = '0' then
  (1) Load pipeline registers
  (2) Instruction Decode
  (3) Instruction Fetch
else
  (1) load PC with abort
      continuation address

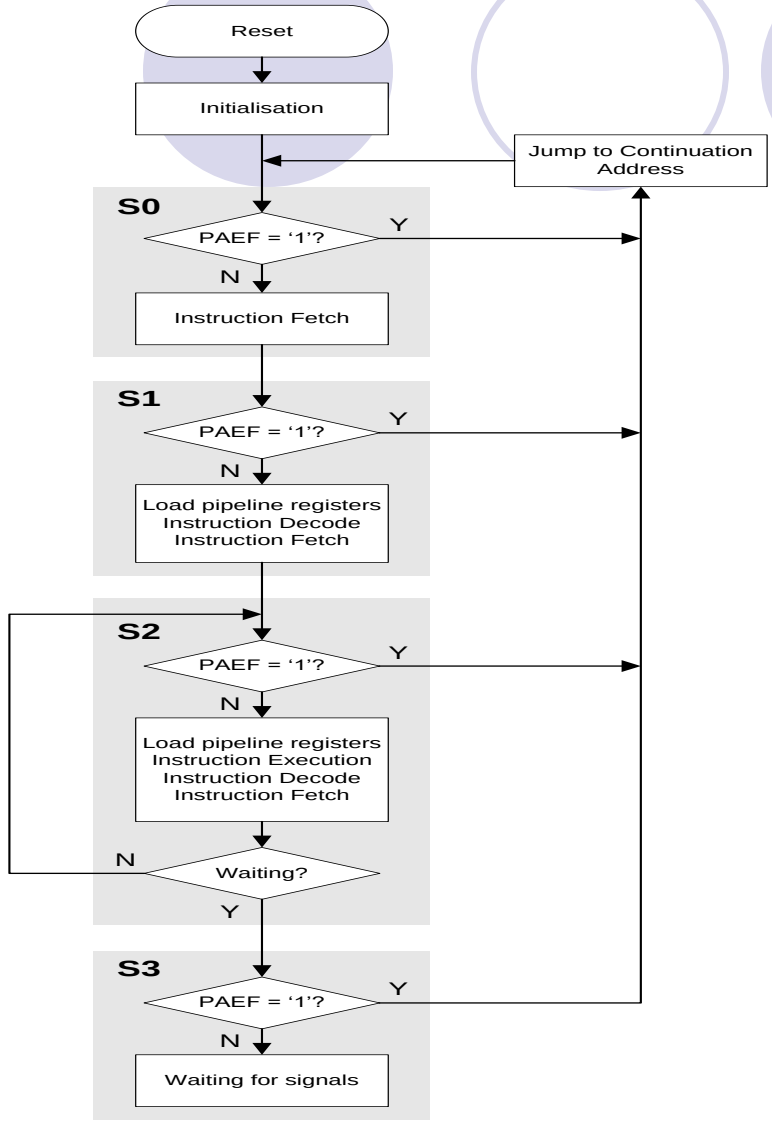
S2
if PAEF = '0' then
  (1) Load pipeline registers
  (2) Instruction Execution
  (3) Instruction Decode
  (4) Instruction Fetch
else
  (1) load PC with abort
      continuation address

S3
if PAEF = '0' then
  (1) Wait for input signals
else
  (1) load PC with abort
      continuation address

State Transition Condition:

a: PAEF = '1'
b: PAEF = '0'
c: PAEF = '0' and pipeline stall detected
d: (PAEF = '1') OR (PAEF = '0' AND
   branching instructions)
e: PAEF = '0' and SAWAIT/CAWAIT/
   TAWAIT/SUSTAIN
f: PAEF = '0' and signal detected
   (exclude CAWAIT Signal 2)
g: (PAEF = '1') OR (PAEF ='0' AND
   CAWAIT Signal 2 detected)

ReMIC – Control Flow



Example Program – Seat Belt Controller

Esterel

```
1: module SEATBELT_CONTROLLER:
2: input key_on, key_off, belt_on, timer_out;
3: output timer_on, alarm_on;

4: loop
5:      await key_on;
6:      emit timer_on;
7:      abort
8:              await timer_out;
9:              sustain alarm_on;
10:     when
11:             case belt_on
12:             case key_off
13:     end abort;
14: end loop

15: end module
```

ReMIC Assembly

```
KEY_ON          &EQU 0  ;KEY_ON = SIP[0]
KEY_OFF         &EQU 1  ;KEY_OFF = SIP[1]
BELT_ON         &EQU 2  ;BELT_ON = SIP[2]
ALARM_ON        &EQU $1 ;ALARM_ON = SOP[0]
TIMER_VAL       &EQU $FFFF

CHK_KEYON       SAWAIT @KEY_ON
                  ABORT @KEY_OFF DONE
                    ABORT @BELT_ON DONE
                      TAWAIT0 #TIMER_VAL
                      SUSTAIN #ALARM_ON
DONE            JMP CHK_KEYON
                &END
```

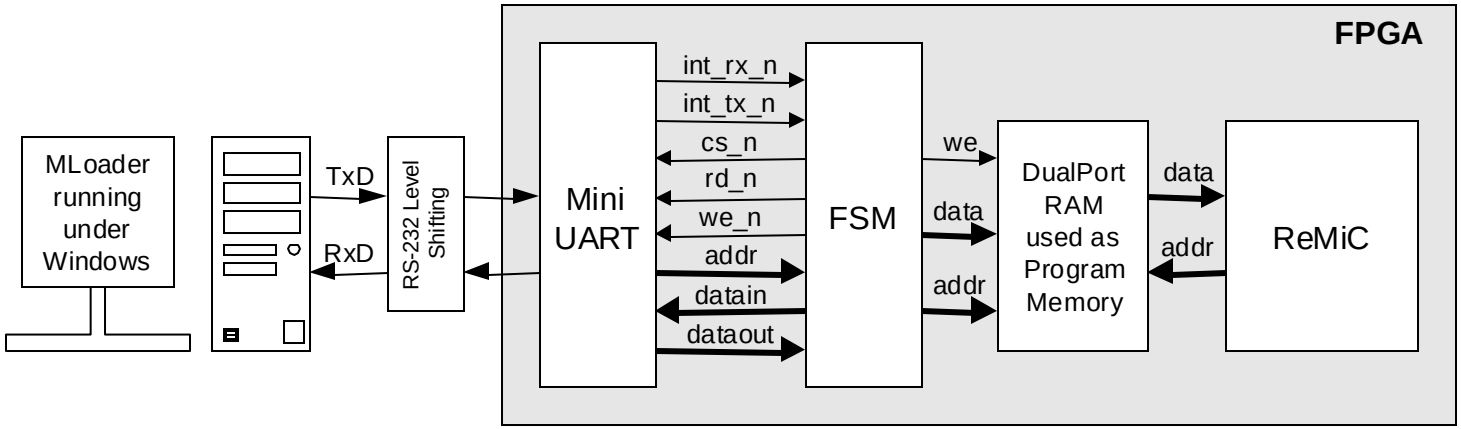
Tools – Macro-Assembler and Instruction Set Simulator

- Assembler directives:
  - Equate (EQU)
  - Origin (ORG)
  - End of source (END)
  - Conditional block (IF,THEN,ELSE, ENDIF)
  - Include (INCLUDE)
  - Macro definitions (MACRO, ENDMACRO)

Tools – Program Loader

Enables independent software development and loading without re-synthesis of hardware

A small FSM communicates with the development system on PC and loads program into PM

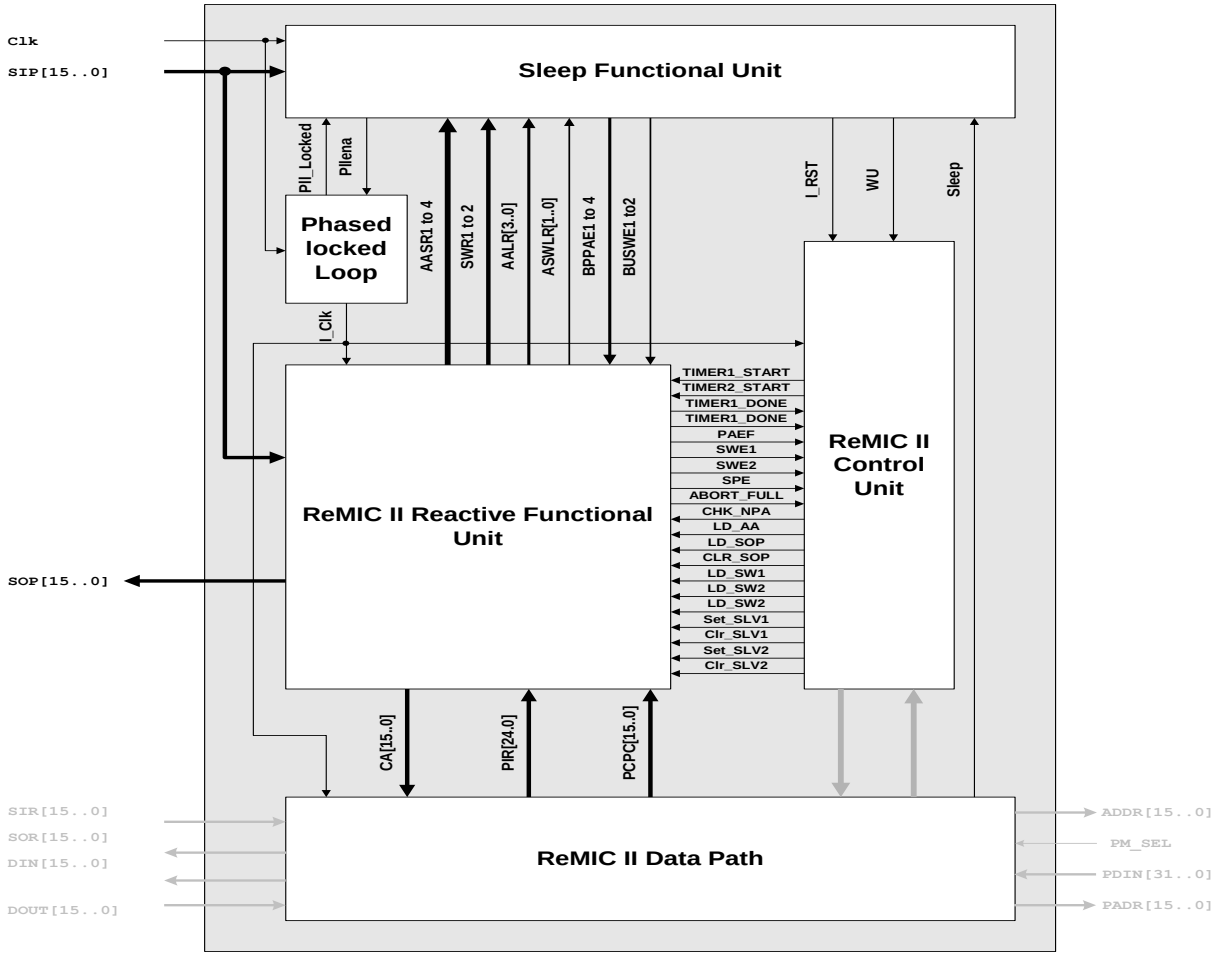


Power-aware ReMIC – ReMIC-PA

Power-aware ReMIC – ReMIC-PA

- Power-optimised datapath
- Phased-locked-loop (PLL) for clock signal generation
- Sleep Functional Unit (SFU) to control PLL
- Additional instructions
- 15% more logic elements to implement
- Substantial power savings on typical data-driven and control-driven applications
- Fast transition from sleep into normal mode
- ReMIC II (with additional instructions to support power consumption reduction)

Power-aware ReMIC – ReMIC-PA



Power-aware ReMIC – ReMIC-PA

Energy efficient instructions

Features	Instruction Syntax	Corresponding Reactive Instruction	Function/Description
Energy-efficient Signal Sustenance	LSUSTAIN *signal(s)*	SUSTAIN	• Bring the processor to the sleep mode and set signal(s) high forever
Energy-efficient Signal Polling	LSAWAIT *signal*	SAWAIT	• Bring the processor to the sleep mode and wait until the specified signal occurs in the environment.
Energy-efficient Conditional Polling	LCAWAIT *signal1, signal2, address*	CAWAIT	• Bring the processor to the sleep mode and wait until either signal1 or signal2 occurs. If signal1 occurs, the processor is restored to the normal mode and executes instruction from consecutive address; or else from the specified address.
Suspend	AWAIT	NONE	• Bring the processor to the sleep mode.

Scheduling Support Unit For Reactive Microprocessor

Outline

- The importance of scheduling in implementation of RTOSs
- Scheduling mechanisms
- Static priority scheduling policy
- Hardware scheduling support unit (SSU) and its programming model
- SSU for ReMIC
- Some performance results

Background

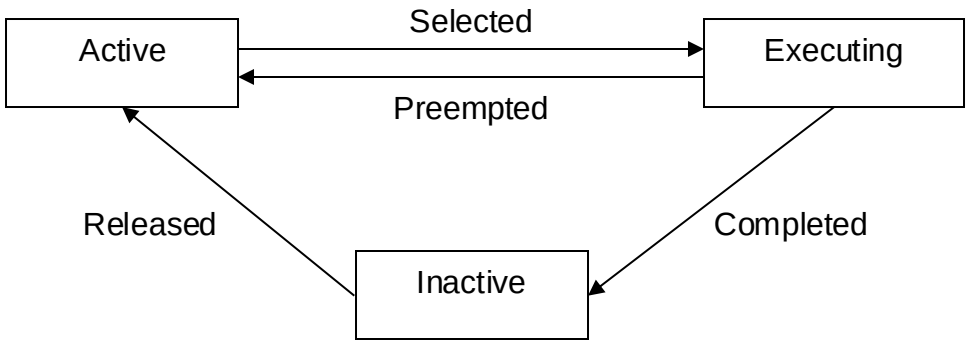
- Original ReMIC microprocessor does not support context switching
- Not suitable for multiple tasks executions
- No queuing of tasks released for execution
- No automatic mechanism for context saving/restoring

- SSU extends original processor to support execution of multiple tasks with different priorities on the same processor
  - tasks are reactive – released, do the processing, produce the outputs, completed/deactivated
- Suitable for concurrent reactive systems with timing constraints

Architectural Framework

- The system consists of a set of reactive tasks that interact with external environment and between themselves
  - using signals and
  - operations on signals (emit, checking for presence, polling, preemption)

- Individual tasks are represented by unique states and taken into those states by the Scheduler

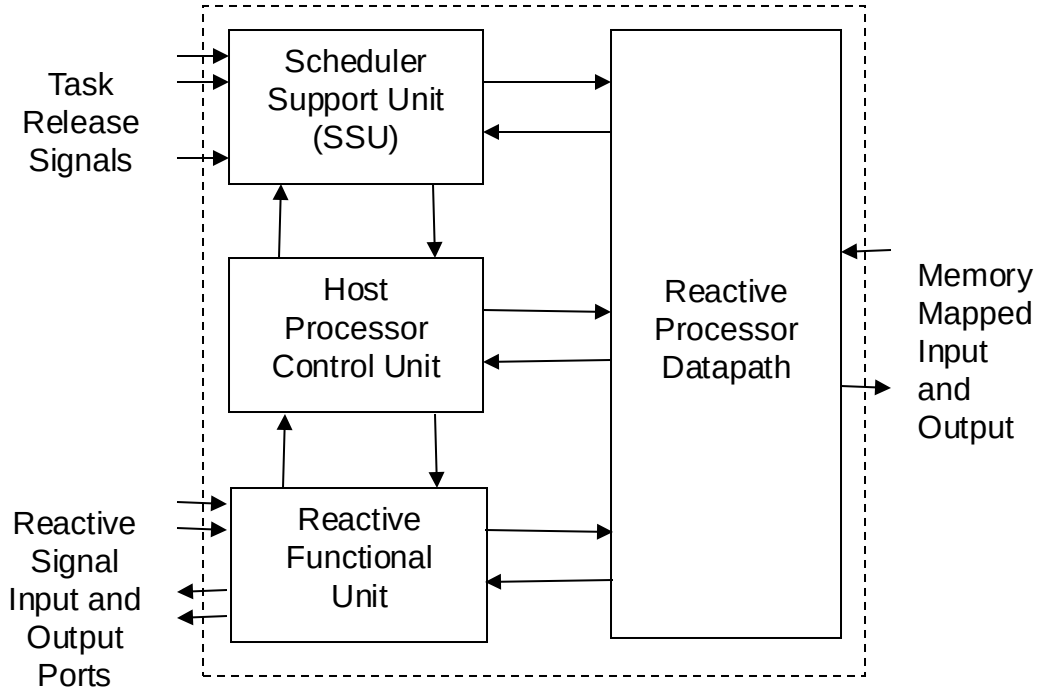


Architectural Framework

- An executing task may be preempted whenever a higher priority task is released
- The SSU resolves priorities of all tasks whenever
  - A new task is released or
  - Currently executing task completes execution
- The highest priority task is selected for execution
- Constraints:
  - The tasks are independent each of the other
  - Priorities are statically assigned (limits applicability to Deadline Monotonic (DM) or Rate Monotonic (RM) scheduling policies
- The events that release tasks arrive asynchronously and the tasks are queued for later execution

Architectural Framework

- One external task release signal associated with each task
- These signals come from sensors, external devices, but also can be generated by timers within the system
- The SSU tracks the state of each task, performs scheduling and context switching
- A task can communicate with the environment through standard ReMIC mechanisms



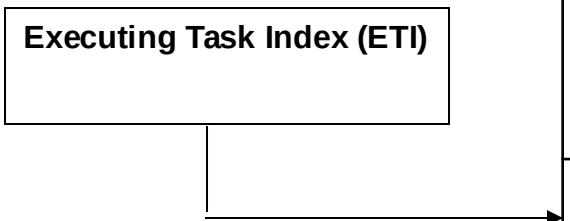
Task Abstraction

- Each task represented with sufficient information for scheduling and context switching
  - Task Descriptor
  - Task Code

- Task Descriptor consists of
  - Task State Indicator TSI (active or non-active)
  - Task Release Signal TSR Code
  - Task Context TC (minimal information to start or resume execution of a task)

Task Abstraction

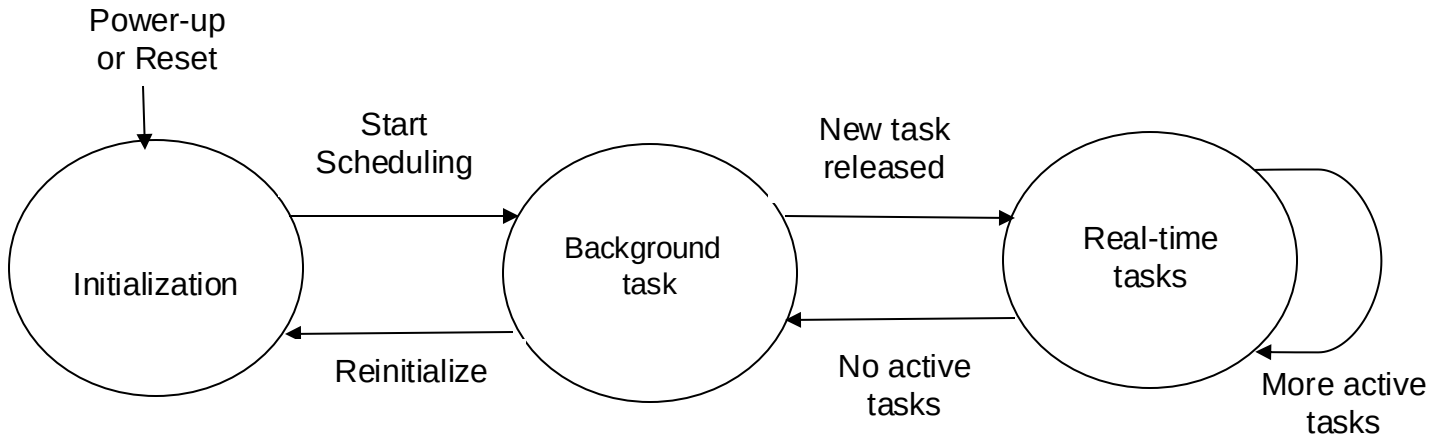
- Tasks are organised in the Task Table where Executing Task Index (ETI) points to the currently executing task



Task State Indicator, TSI (1 bit)	Task Release Signal, TRS (m bits)	Task Context TC (48 bits)
		Task 1 (highest priority)
		Task 2
		Task N (Background task)

Task Execution Model

- Whenever there is no real-time task, a background task is executed



Scheduling Support Instructions

- Additional instructions implemented to control SSU
  - To initialise Task Table
  - Enabling and suspending (stopping) and resuming scheduling
  - Specification of the context in which the task will be reactivated or resumed
- Instructions integrated into ReMIC instruction set

Scheduling Support Instructions

Action	Command Format	Description
Initialize task	**InitTask** Sig, Context	The task descriptor is created in the next free slot of the task-table; when the task is released by signal **Sig** it starts from the specified **Context**.
Start scheduling	**Enable** SSU	The SSU is enabled and tasks can be released by the occurrence of events on signal lines; Executing Task Index is initialized to the lowest priority task (background task).
Stop scheduling	**Stop** SSU	The SSU is disabled and tasks are not released by occurrence of signals on signal lines – the host processor continues executing the background task.
Resume scheduling	**Resume** SSU	The SSU is enabled and tasks can be released by occurrence of events on signal lines; Executing Task Index remains unchanged from the value it had when scheduling stopped.
Task completed	**CompleteTask** Context	The task indicates its completion to the SSU and specifies the context from which it will start after its next release.

Multi-tasking Programming Model

```
; Initialize SSU (task-table)
InitTask s0, Context0
InitTask s1, Context1
.
.
.
InitTask sk, Contextk
Enable SSU
```

```
; Task specific code

while (true) do
BackgroundTask: ; can use stop and resume SSU
...
end while;

Task1:
...

CompleteTask Context1; end of task1

Task2:
...
CompleteTask Context2; end of task2

TaskK:
...
CompleteTask ContextK; end of taskK
```

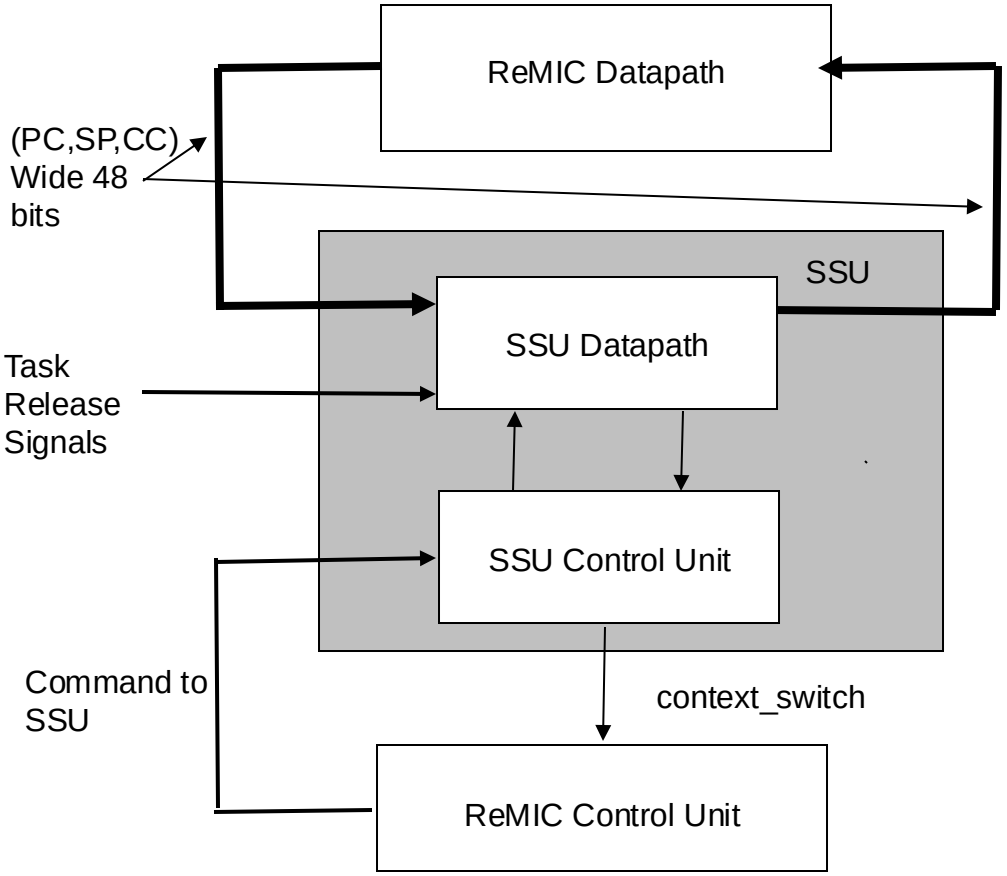
Scheduling Support Unit - ReMIC

- Task Context – defined with PC, SP (SR), CC
- For saving/restoring task context in a single clock cycle a specialised datapath provided (added to original ReMIC datapath)
- Storage for Task Descriptors split into two parts

- TSI and TSR implemented using flip/flops
- TC part implemented in SRAM of FPGA

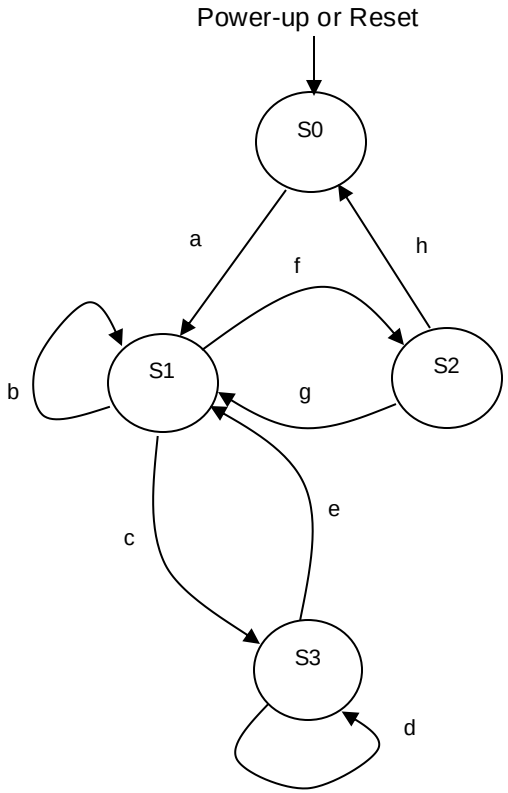
Scheduling Support Unit - ReMIC

- SSU has priority resolution circuitry that compares priority of newly released task with the currently executing task
- It controls scheduling by stalling the main processor and providing consistency of its pipeline
- Executes commands received from the main processor



Scheduling Support Unit for ReMIC

- Restoration or change of processor context is sole decision of the Scheduler and is controlled by cotext_switch signal from SSU
- Main control unit: the oldest instruction in the pipeline corresponds to the context of the current executing task



FSM States:
S0 – SSU Initialization
S1 – Background task – SSU enabled
S2 – Background task – SSU disabled
S3 – Executing real-time task

FSM Transitions:

a – SSU enabled (start scheduling)
b – Task not released
c – Task released; context_switch = 1
d – Task completed or released; not all tasks inactive; context_switch = 1
e – Task completed; all tasks inactive; context_switch = 1
f – SSU disabled
g – SSU enabled
h – Reinitialize task table

ReMIC with or without the SSU – Performance Comparisons

Feature	Original core	Modified core additional resources			
		2 tasks	4 tasks	8 tasks	16 tasks
Additional Logic elements	1472	1562	1583	1640	1703
Additional logic elements (% increase)		6.1	7.5	11.4	15.7
Additional SRAM Memory (bits)	8x16	6x16	12x16	24x16	48x16
Clock frequency MHz)	65	60	60	60	60

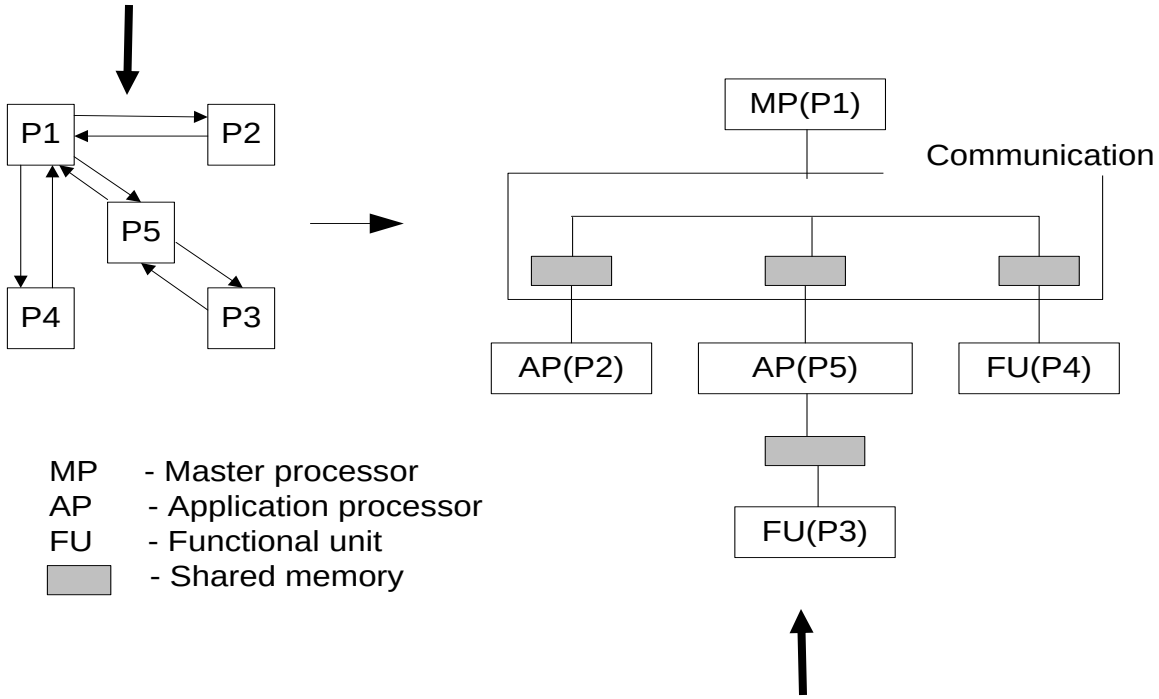
HiDRA – HybriD Reactive Architecture

HiDRA – HybriD Reactive Architecture

- Combination of synchronous models of computation with broadcast and data-flow
- Globally asynchronous locally synchronous (GALS) behaviours
- Concurrency on single processor, hardware-implemented modules, multiple-processor and any of their combination
- Direct mapping of system-level description (modified Esterel or SystemC) onto the new architecture

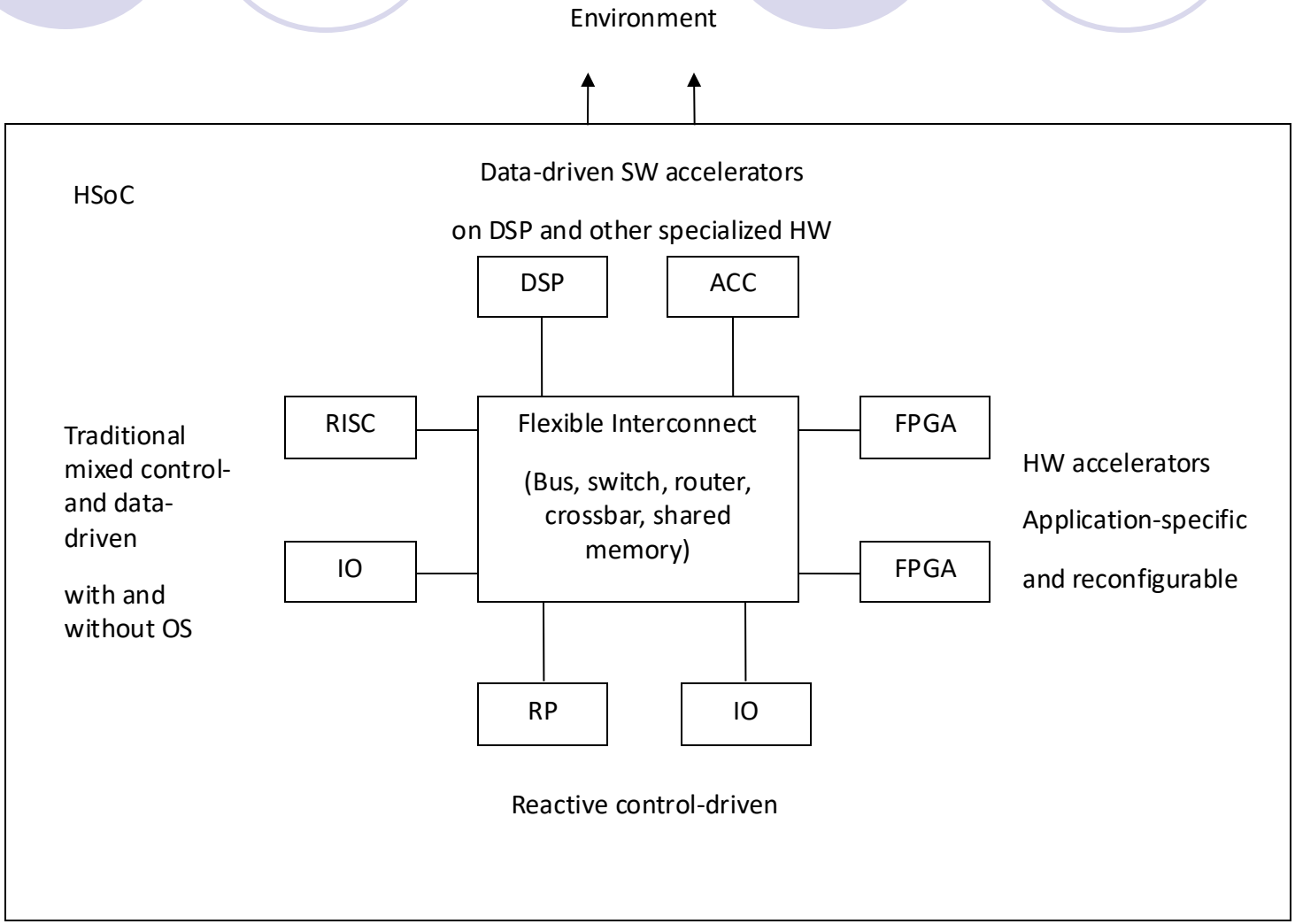
HiDRA – Mapping an Application onto HiDRA Architecture

Application - set of communicating behaviours (processes)

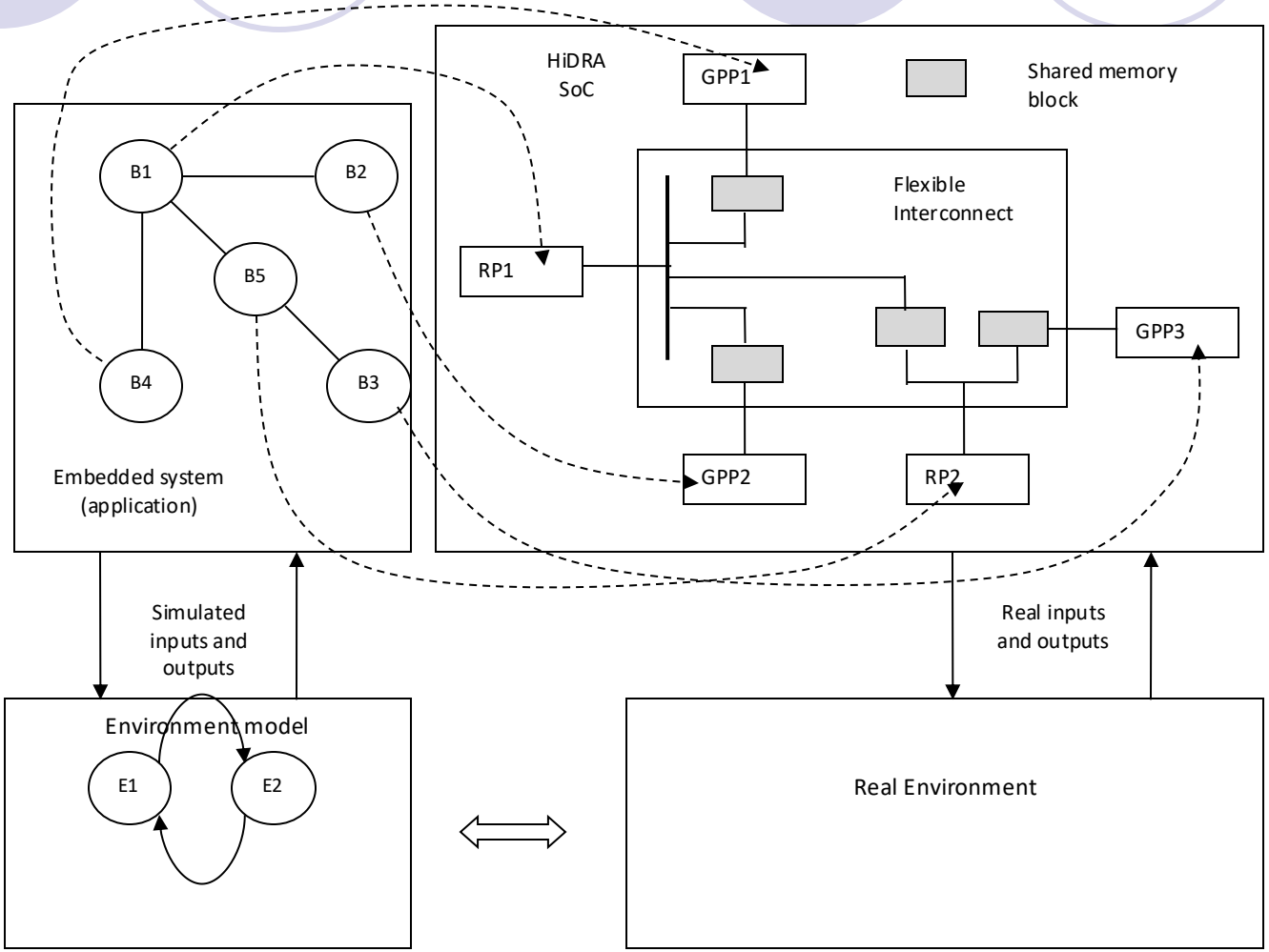


Behaviour allocation on hardware and software components (MP, AP, FU)

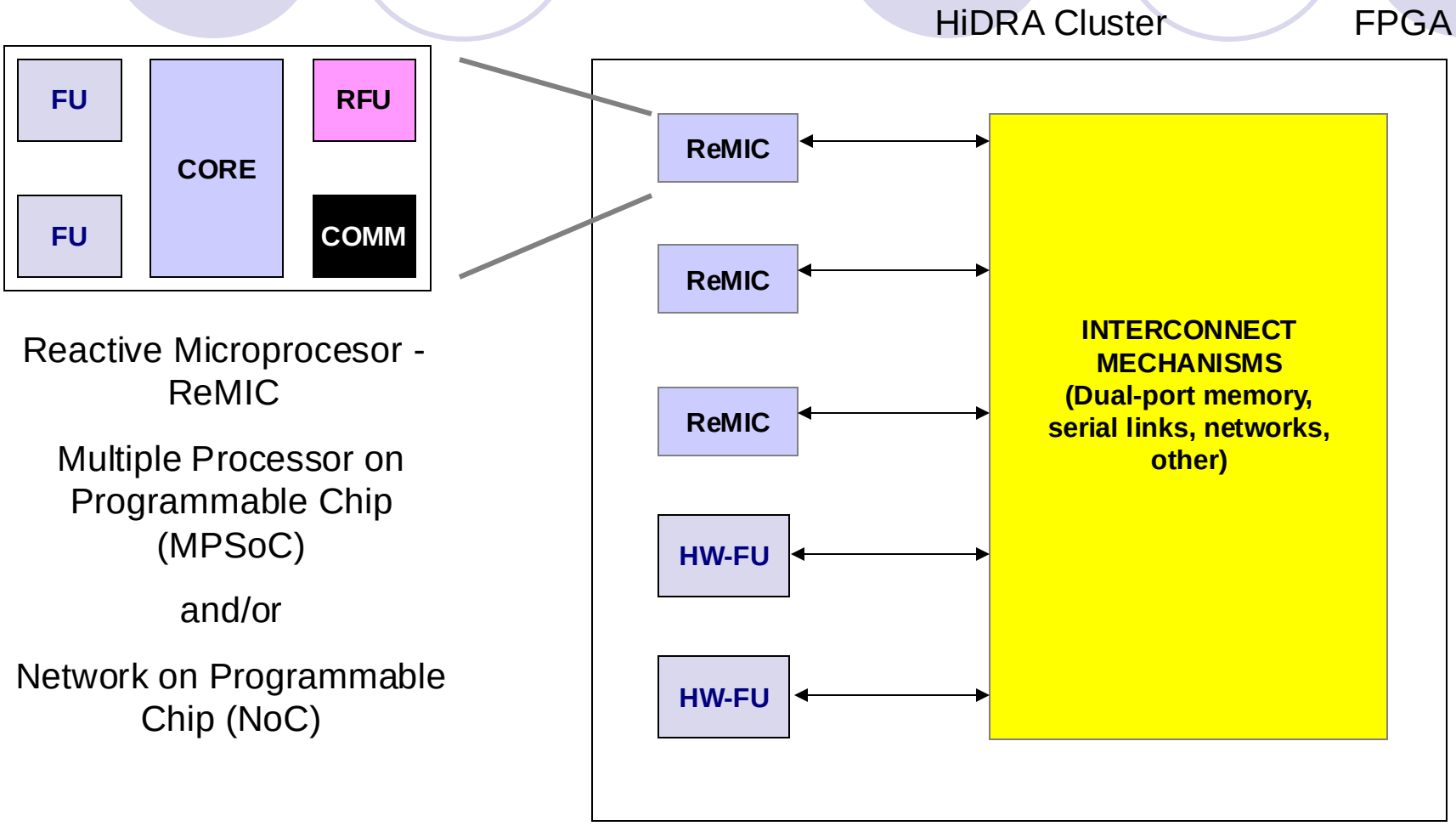
Heterogeneous Systems on Chip (HSoC)



Mapping an Application to HiDRA



HiDRA – HybriD Reactive Architecture



Reactive Microprocesor - ReMIC

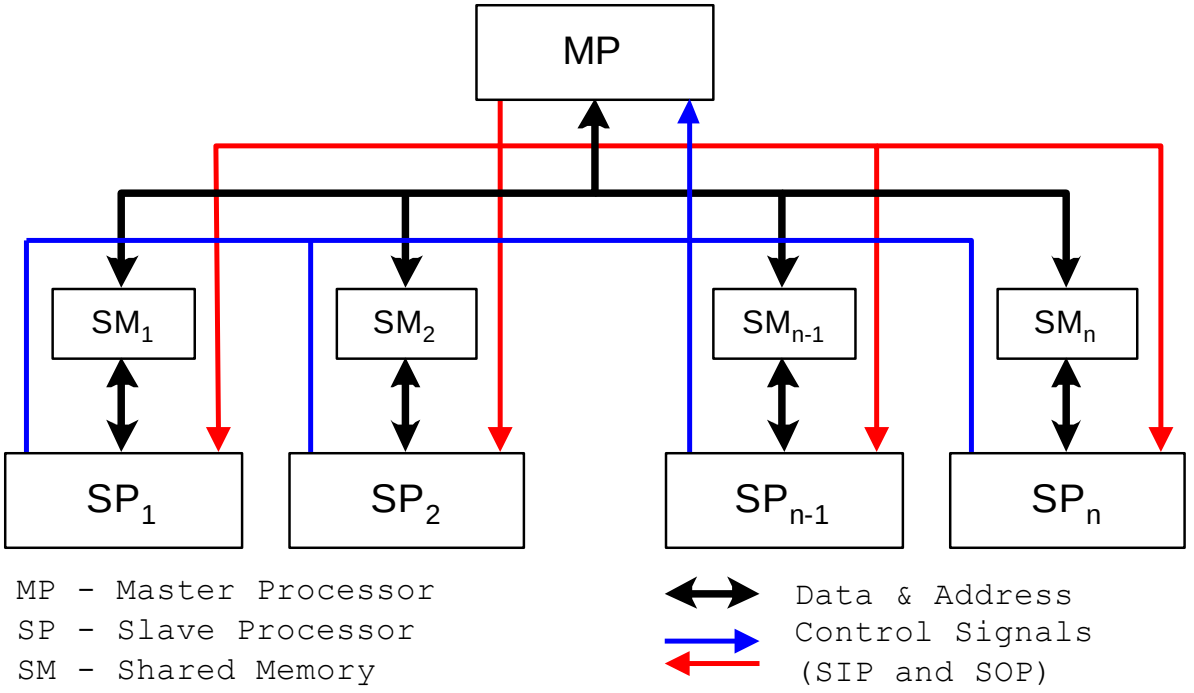
Multiple Processor on Programmable Chip (MPSoC)

and/or

Network on Programmable Chip (NoC)

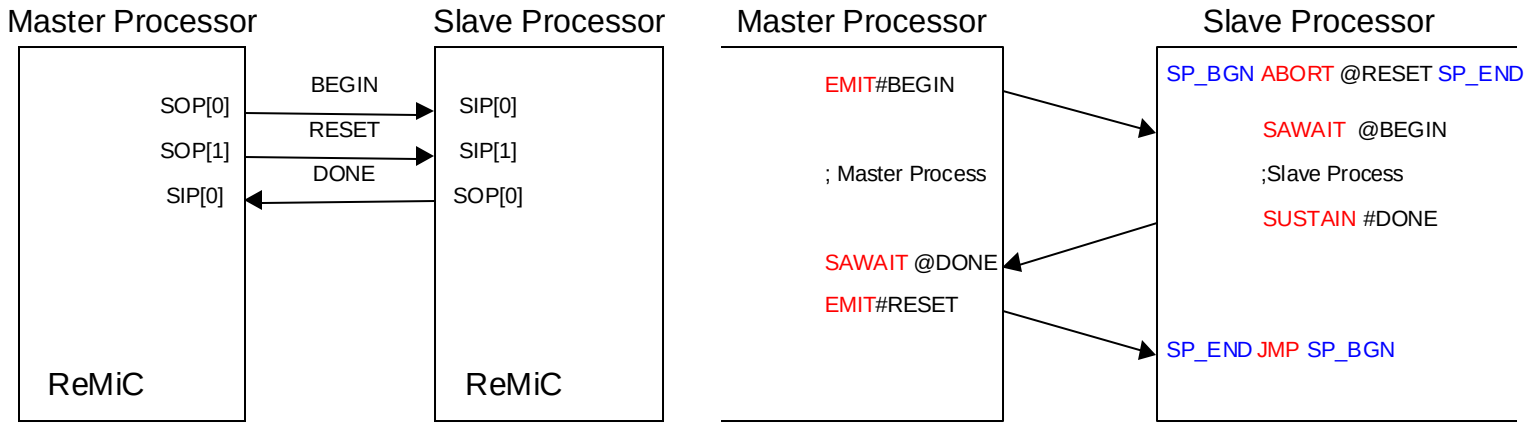
HiDRA – General Architecture

- MP (Master Processor) – responsible for system initialisation and coordination of concurrent activities
- SP (Slave Processor) – implement concurrent activities



HiDRA – Low-level Synchronisation Mechanism

- Handshaking between MP and SP

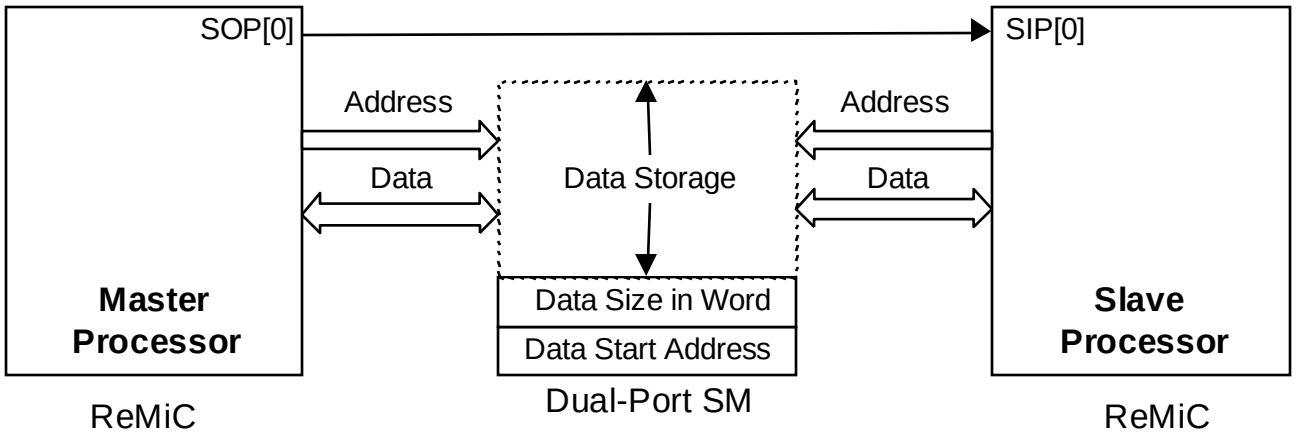


(a) Hardware Connection

(b) Software Implementation

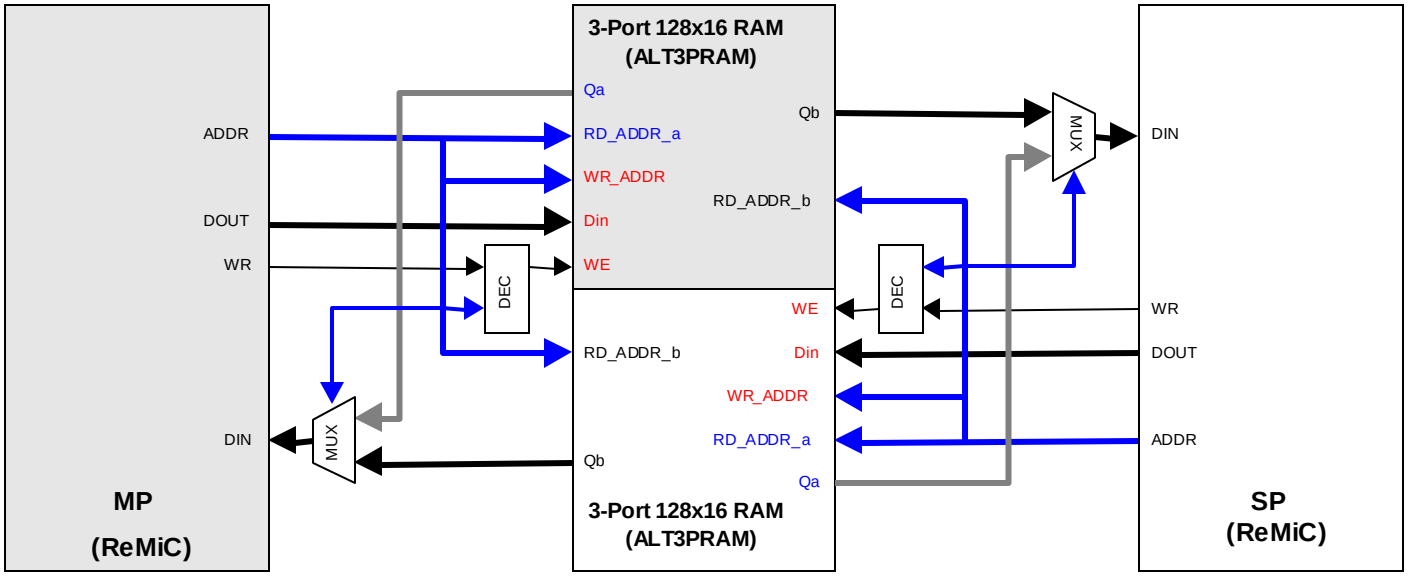
HiDRA – Low-level Communication Mechanism

- Messages can be transferred using small shared memory blocks
- This supports exchange of “valued” signals in Esterel sense



HiDRA – Low-level Communication Mechanism

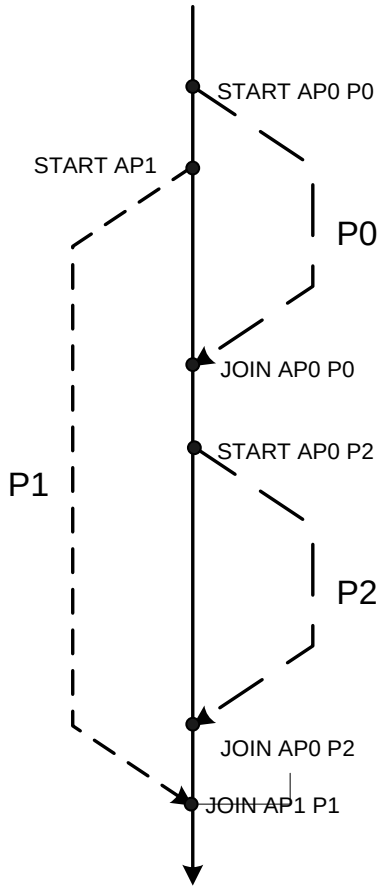
- Implementation on Altera FPGAs
- Using standard 3-port memory blocks (2 read and 1 write port)
- Memory can be placed at any place in address space – can be different for SP and MP



HiDRA Programming Support – Concurrent Reactive Assembly Language (CRAL)

Concurrency		
Software process activation	**START** processorID processID,…	
Wait on software process termination	**JOIN** processorID, processID	
Hardware process activation	**STARTFU** FUID, command	
Wait on hardware process termination	**WAITFU** FUID	
Communication		
Send message to software process	**SEND**(message, processID)	
Receive message from software process	**RECEIVE**(message, processID)	
Send message to hardware process	**SENDF**(message, FUID)	
Receive message from hardware process	**RECEIVEF**(message, FUID)	
Parallelism control commands		
Main program beginning	**&MAIN_BEGIN**	
Main program end	**&MAIN_END**	
Slave program beginning	**&SPR_BEGIN**	
Slave program end	**&SPR_END**	

HiDRA Programming Support – CRAL Usage Example



Master Processor _____

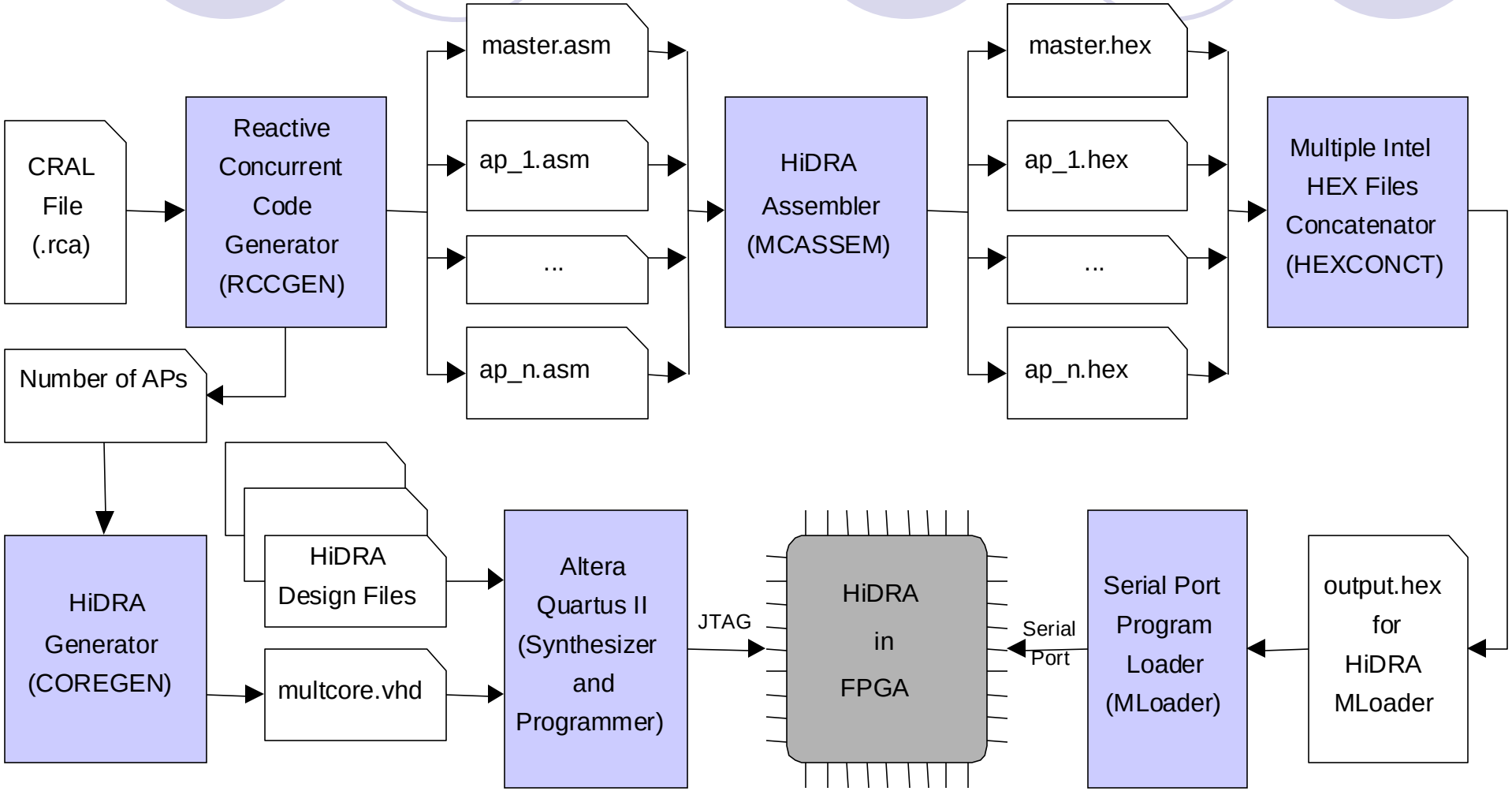
Slave Processor 1 — — —

Slave Processor 2 _ _ _ _ _ .

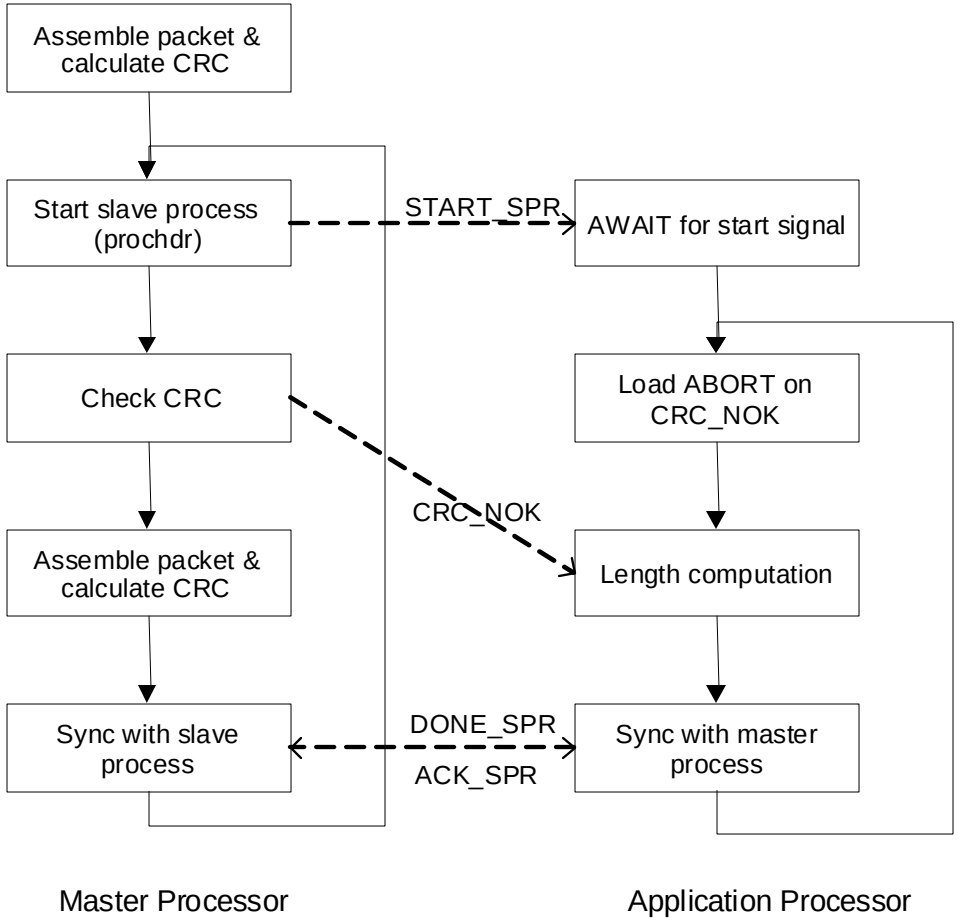
Reactive Concurrent Program:

```
&MASTER_BEGIN
…
START AP0 P0
…
START AP1 P1
…
JOIN AP0 P0
…
START AP0 P2
…
JOIN AP0 P2
…
JOIN AP1 P1
…
&MASTER_END
```

HiDRA – System Design Flow

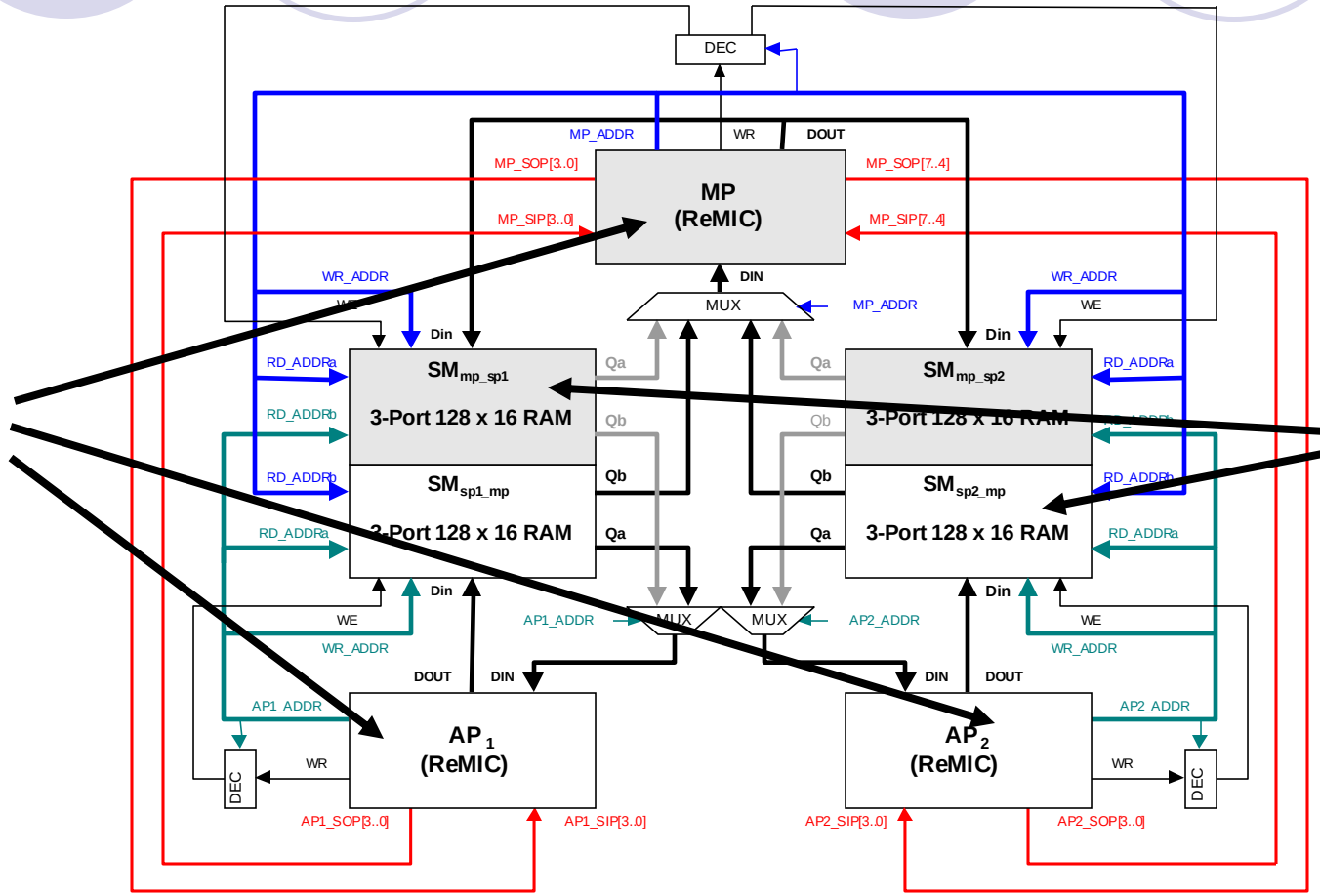


HiDRA Example Application – Protocol Stack Implementation Conceptual Solution



HiDRA Example Implementation

ReMIC Cores



Shared Memories

Synchronisation signals

HiDRA Example Application – Protocol Stack Implementation Results Comparison

Implementation	Data memory (bytes)	Code size (bytes)		Exec. Time (cycles)	
		MP	SP	MP	SP
HiDRA	64	160	32	810	30
HiDRA (total)	64	192		1113	
MIPS (1 task)	160	1008		4283	
MIPS (3 tasks)	352	1632		4161	

Documents to Read

Dong Hui and Zoran Salcic

“MiCORE – A Customisable Microprocessor Core”

“ReMiC – A Customisable Reactive Microprocessor Core”

University of Auckland

Department of Electrical and Computer Engineering

Embedded Systems Research Group Internal Report

July 2004, available on Canvas