

**Volume**

**1.1**

UNIVERSITY OF TEHRAN AND CINI CYBERSECURITY NATIONAL LABORATORY

---

**SAYAC**

*Simple Architecture Yet Ample Circuitry*

UNIVERSITY OF TEHRAN AND CINI CYBERSECURITY NATIONAL LABORATORY

# SAYAC Reference Manual

---

Version 1.1

Embedded Core Guide

Under the supervision of Prof. Zain Navabi<sup>1</sup>, Prof. Paolo Prinetto<sup>2</sup>

<sup>1</sup>Worcester Polytechnic Institute, <sup>2</sup> CINI Cybersecurity National Laboratory

[navabi@wpi.edu](mailto:navabi@wpi.edu), [paolo.prinetto@polito.it](mailto:paolo.prinetto@polito.it)

August 2022

---

# Proprietary Notice

The present document offers information subject to the terms and conditions described here- inafter. All rights are reserved to CINI Cybersecurity National Laboratory and University of Tehran. Copy- right and related rights are licensed under the GNU Lesser General Public License, Version 3.0; you may not use this file except in compliance with the License. You may obtain a copy of the License at <https://www.gnu.org/licenses/lgpl-3.0.txt>. Unless required by applicable law or agreed to in writing, software, hardware and materials distributed under this License is distributed on an “as is” basis, without warranties or conditions of any kind, either express or implied. See the License for the specific language governing permissions and limitations under the License. The authors reserve the possibility to change the content and information described in this document and to update such information at any time, without notice. Despite the attention that has been taken in preparing this document, typographical errors, error or omissions may have occurred.

## Authors

Alireza NHVI (PhD candidate, University of Tehran) [alireza\\_nahvy@ut.ac.ir](mailto:alireza_nahvy@ut.ac.ir)  
Hanieh TOOTOONCHI-ASL (Master Student, University of Tehran) [h.totonchi@ut.ac.ir](mailto:h.totonchi@ut.ac.ir)  
Hasti KOWSAR (Student, University of Tehran) [hasti.kowsar@ut.ac.ir](mailto:hasti.kowsar@ut.ac.ir)  
Sepideh KHEIROLLAHI (Student, University of Tehran) <mailto:s.kheirollahi@ut.ac.ir>  
Katayoon BASHARKHAH (PhD candidate, University of Tehran) [basharkhah.kt96@ut.ac.ir](mailto:basharkhah.kt96@ut.ac.ir)  
Zainalabedin NAVABI (Full Professor, University of Tehran) [navabi@ut.ac.ir](mailto:navabi@ut.ac.ir)  
Paolo PRINETTO (Director, CINI Cybersecurity National Lab) [paolo.prinetto@polito.it](mailto:paolo.prinetto@polito.it)

Disclaimer THIS IS THE ROUGH DRAFT OF SAYAC MANUAL WITH THE DESCRIPTION OF ITS HARDWARE AND BY NO MEANS IS REPRESENTED. THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN “AS IS” BASIS AND ITS AUTHORS DISCLAIM ALL WARRANTIES, EXPRESS, OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PURPOSE. THE SOFTWARE IS PROVIDED TO YOU “AS IS” AND WE MAKE NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER WITH RESPECT TO ITS FUNCTIONALITY, OPERABILITY, OR USE, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PURPOSE, OR INFRINGEMENT. WE EXPRESSLY DISCLAIM ANY LIABILITY WHATSOEVER FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR SPECIAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOSS REVENUES, LOST PROFITS, LOSSES RESULTING FROM BUSINESS INTERRUPTION OR LOSS OF DATA, REGARDLESS OF THE FORM OF ACTION OR LEGAL THERE- UNDER WHICH THE LIABILITY MAY BE ASSERTED, EVEN IF ADVISED OF THE POSSIBILITY LIKELI- HOOD OF SUCH DAMAGES.

## Acknowledgments

This work is the result of a fruitful collaboration and friendship between the CINI Cybersecurity National Laboratory and the University of Tehran, whose inspirers are certainly Prof. Paolo Prinetto and Prof. Zainalabedin Navabi

---

# Table of Contents

1. An Overview .....	1
1-1 Features.....	1
1-2 Embedded View.....	1
2. Programmer's Model .....	3
2-1 Privilege Levels .....	3
2-2 Instruction Set Summary .....	3
2-3 Register Summary.....	6
2-4 Exceptions.....	7
2-5 Interrupts .....	7
3. Memory Model.....	9
3-1 Memory Map .....	9
3-2 Caches .....	10
3-3 Memory Protection Unit.....	13
4. SAYAC Core Architecture .....	16
4-1 Overview .....	16
4-2 Datapath.....	17
4-3 Controller.....	29
5. Exception and Interrupts .....	31
5-1 Exception .....	31
5-2 Interrupt.....	33
6. Interfaces .....	35
6-1 System Bus .....	35
6-2 Bus components.....	35
6-3 Cache Interface Unit.....	37
7. System User Interface .....	39
7-1 SUI commands.....	39
7-2 Example System .....	43
7-3 Example System Command Line Results .....	46
Index.....	49

---

## 1. An Overview

**T**he SAYAC processor is a Simple Architecture Yet Ample Circuitry. This is a RISC-like open-source academic processor that was originally designed to support educational processor hardware architecture and implementation.

However, the simple and ample architecture of this processor provides increased value enabling the researchers to devise new research directions in design, test, reliability, high-level modeling, and security. SAYAC is a 16-bit processor with a 16-bit address bus and a 16-bit bi-directional data bus. The processor has standard memory accessing handshaking signals, and separate IO read and write lines. The latter arrangement simplifies IO processing in small applications that do not require a complicated bussing structure for separation of memory from IO. Despite the small range of instruction width, all necessary instructions for executing different applications are covered. This is done by using the concept of shadow instructions.

### 1-1 Features

- RISC architecture
- General and Special purpose registers: 16 16-bit registers in a register file
- External interrupts
- A memory protection unit
- An HDL-Based virtual tester
- Test Supports

### 1-2 Embedded View

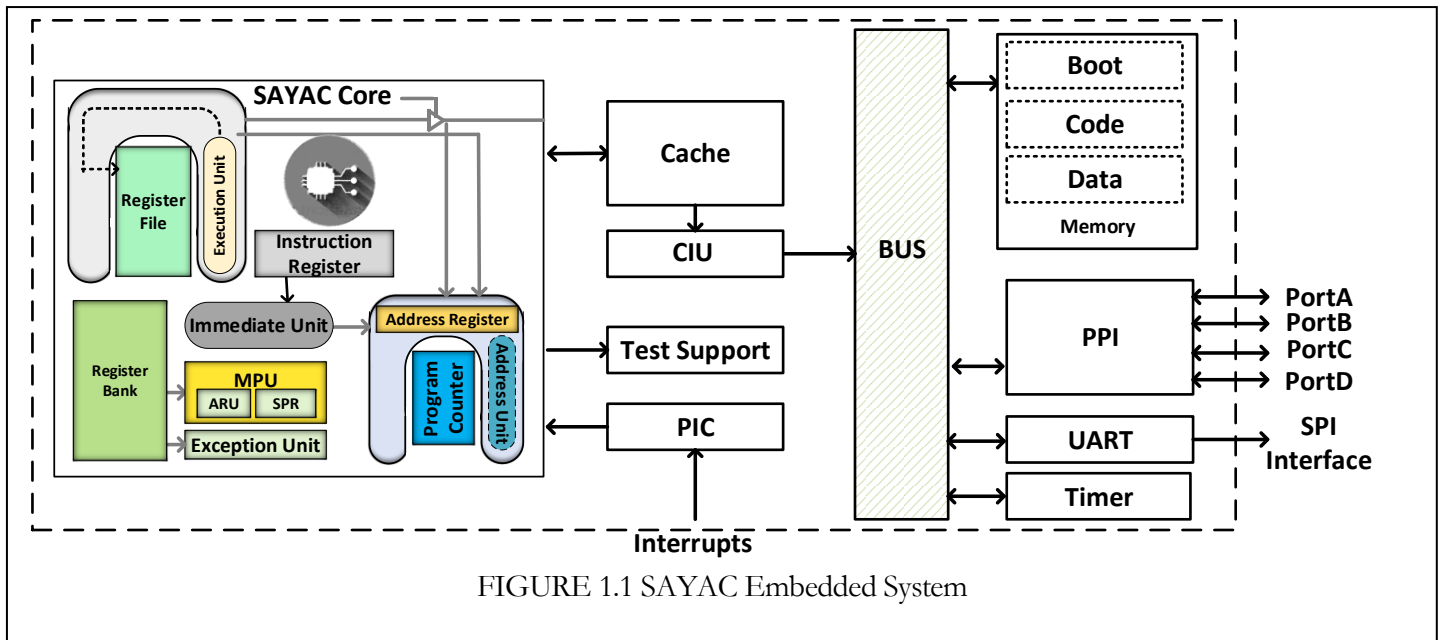
FIGURE 1.1 shows an overall view of the SAYAC processor in an embedded system. Based on this figure, the complete system component list comprises:

- SAYAC Processor Core:
  - **MPU:** Memory Protection Unit including 4 main memory regions.
  - Interrupt and Exception units
  - Register bank for control state registers
  - Register file as general-purpose registers



NOTE

For more information on SAYAC Processor core, see [Chapter 4](#).



- Memory Components:
  - Cache unit that is tightly coupled with the processor core
  - Memory protection unit

NOTE For more information on Memory components, see [Chapter 3](#).

- Interrupt Components:
  - **PIC:** Programmable Interrupt Controller

NOTE This component is not implemented in this version and will be added in the future revisions.

- Interfaces:
  - **BUS:** Bus interface unit
  - **CIU:** Cache Interface Unit

NOTE For more information on Interfaces, see [Chapter 6](#).

## 2. Programmer's Model

This chapter describes the SAYAC processor register set, modes of operation, and provides information on programming the SAYAC processor.

### 2-1 Privilege Levels

SAYAC processor has two software privilege levels (in increasing order of capability): user-mode (U-mode) and machine mode (M-mode). The processor can run in only one of the privilege modes at a time. Privilege levels will be encoded in the status register  $R_7(15)$  based on the TABLE I.

Privilege levels are used to provide protection between different components of the software stack and attempts to perform operations not permitted by the current privilege mode cause an exception to be raised. The machine level has the highest privilege and is the only mandatory privilege level for SAYAC platform. Code run in machine mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on SAYAC. User mode (U-mode) is intended for conventional application). All hardware implementations must provide M-mode, as this is the only mode that has unfettered access to the whole machine.

**TABLE I: SAYAC Privilege Levels**

Level	Encoding	Name	Abbreviation
0	00	Machine	M
1	01	User/Application	U

### 2-2 Instruction Set Summary

SAYAC supports 32 instructions. Four most significant bits of instruction word is reserved as Operation Code (opcode). To be able to cover more instructions, for some formats this opcode will be extended up to 8-bits. The instructions of the SAYAC processor can be categorized in three main groups: R-type (Register type), I-type (Immediate type) and S-type (System type).

FIGURE 2.1 shows various formats used in SAYAC instruction set. Instructions operate on registers of the register-file. The register-file has a destination address (rd) and two source addresses (rs1 and rs2). All the instructions except the system type instructions have a destination register at the four least significant bits of the instruction word as shown in FIGURE 2.1.

R-type instructions can include one or two of source addresses. Format shown in FIGURE 2.1 (a) uses two 4-bit source register addresses and has a 4-bit destination address. Logical and arithmetic instructions use this format. As an example, the add instruction (ADR) performs an add on the contents of the register in the register-file structure addressed by  $r_{s1}$  with contents of another register in this structure -addressed by  $r_{s2}$  and puts the result in register-file register addressed by  $r_d$ . Format shown in FIGURE 2.1 (b) is another R-type instruction that have one source address and one destination address. For these instructions the opcode is extended up to 8-bits and they will mostly be used for memory instructions (load and store), one-operand logical instructions (like NOT) and control flow instructions (like compare).

For instructions that we refer to as immediate, as shown in FIGURE 2.1 (c), one of their sources is taken from certain bits of the Instruction Register (IR). The bit width of immediate varies between 5 to 8 bits for different instructions; 6 bits for JMI and CMI instruction, 8 bits for logical and arithmetic instructions and 5 bits for SHI instructions.

System instructions all have the fixed opcode of 0xF0 and can support up to 256 system instructions. In the current version of the SAYAC processor, one system instruction, Machine Environment Call (MEC) has been defined.

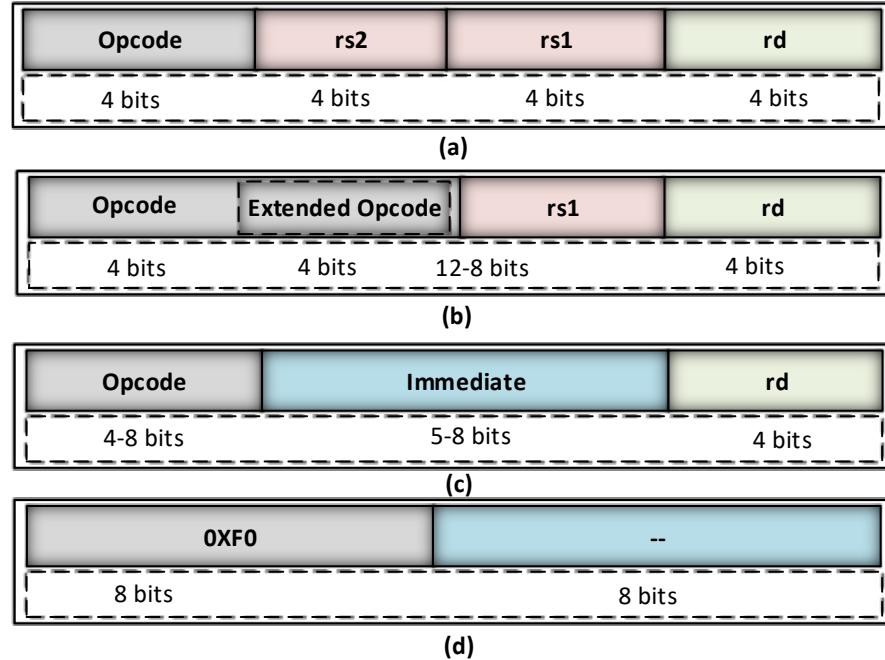


FIGURE 2.1 SAYAC instruction formats (a) and (b) R-type, (c) I-type and (d) S-type



**TABLE II:** SAYAC Instruction Set

[15:12]	[11:10]	[9]	[8]	[7:4]	[3:0]	Instruction	Notation
0000	Reserved						
0001	Reserved						
0010	00	0	0	rs1	rd	LDR	$R_f(rd) \leftarrow \text{MEM}( R_f(rs1) )$
			1	rs1	rd	LIR	$R_f(rd) \leftarrow \text{IO}( R_f(rs1) )$
		1	0			LDB	$R_b(rd) \leftarrow \text{MEM}( R_f(rs1) )$
			1			LIB	$R_f(rd) \leftarrow \text{IO}( R_b(5) + R_f(rs1) )$
		01	0			0	STR
			1	rs1	rd	SIR	$\text{IO}( R_f(rd) ) \leftarrow R_f(rs1)$
		1	0			STB	$\text{MEM}( R_b(3) + R_f(rd) ) \leftarrow R_f(rs1)$
			1			SIB	$\text{IO}( R_b(5) + R_f(rd) ) \leftarrow R_f(rs1)$
	10	s	0	rs1	rd	JMR	$\text{PC} \leftarrow \text{PC} + R_f(rs1)$ $R_f(rd) \leftarrow \text{PC} + 1$ , if s=1
			1	rs1	rd	JMB	$\text{PC} \leftarrow \text{IO}( R_b(rd) + rs1 )$ $R_f(15) \leftarrow \text{IO}( R_b(rd) )$ , if s=1
	11	imm			rd	JMI	$\text{PC} \leftarrow \text{PC} + \text{SE''imm''}$ $R_f(rd) \leftarrow \text{PC} + 1$
0011	rs2			rs1	rd	ANR	$R_f(rd) \leftarrow R_f(rs1) \text{ AND } R_f(rs2)$
0100	imm				rd	ANI	$R_f(rd) \leftarrow R_f(rd) \text{ AND USE''imm''}$
0101	imm				rd	MSI	$R_f(rd) \leftarrow \text{SE''imm''}$
0110	imm				rd	MHI	$R_f(rd) \text{ 'MSB} \leftarrow \text{'imm''}$
0111	rs2			rs1	rd	SIR	$R_f(rd) \leftarrow R_f(rs1) \text{ LS}_{\pm} R_f(rs2)$
1000	rs2			rs1	rd	SAR	$R_f(rd) \leftarrow R_f(rs1) \text{ AS}_{\pm} R_f(rs2)$
1001	rs2			rs1	rd	ADR	$R_f(rd) \leftarrow R_f(rs1) + R_f(rs2)$
1010	rs2			rs1	rd	SUR	$R_f(rd) \leftarrow R_f(rs1) - R_f(rs2)$
1011	imm				rd	ADI	$R_f(rd) \leftarrow R_f(rd) + \text{SE''imm''}$
1100	imm				rd	SUI	$R_f(rd) \leftarrow R_f(rd) - \text{SE''imm''}$
1101	rs2			rs1	rd	MUL	$R_f(rd) \leftarrow R_f(rs1) \times R_f(rs2)$ 'LSB $R_f(rd+1) \leftarrow R_f(rs1) \times R_f(rs2)$ 'MSB
1110	rs2			rs1	rd	DIV	$R_f(rd) \leftarrow R_f(rs1) \div R_f(rs2)$ 'Quo $R_f(rd+1) \leftarrow R_f(rs1) \bmod R_f(rs2)$ 'Rem
1111	00	0	0	0000	0000	MEC	PRV $\leftarrow$ U-Mode EnvironmentCallException = 1
			1	rs1	rd	CMR	flags $\leftarrow$ Cmp( $R_f(rs1)$ , $R_f(rd)$ )
		1	imm		rd	CMI	flags $\leftarrow$ Cmp( $R_f(rd)$ , SE''imm'' )
	01	0	flags interpretation bits		rd	BRC	$\text{PC} \leftarrow R_f(rd)$ if flag
		1	flags interpretation bits		rd	BRR	$\text{PC} \leftarrow \text{PC} + R_f(rd)$ if flag
	10	0	shim		rd	SHI	$R_f(rd) \leftarrow R_f(rd) \text{ LS}_{\pm} \text{'shim''}$
		1	shim		rd		$R_f(rd) \leftarrow R_f(rd) \text{ AS}_{\pm} \text{'shim''}$
	11	0	0	rs1	rd	NTR	$R_f(rd) \leftarrow 1s\text{Comp}( R_f(rs1) )$
			1		rd		$R_f(rd) \leftarrow 2s\text{Comp}( R_f(rs1) )$
		1	0	--	rd	NTD	$R_f(rd) \leftarrow 1s\text{Comp}( R_f(rd) )$
		1	--	rd		$R_f(rd) \leftarrow 2s\text{Comp}( R_f(rd) )$	

TABLE II shows the list of all instructions supported by the SAYAC processor.

#### Legenda of TABLE II

$r_d$  → destination register  
 $r_{s1}$  → Source register 1  
 $r_{s2}$  → Source register 2  
**imm** → immediate  
**shim** → Sign-and-magnitude 5-bit immediate shift amount  
 $R_f(r_d)$  → Rf content pointed by rsd  
**MEM [adr]** → Memory addressed by adr  
**IO (loc)** → IO device addressed by loc  
**SE”imm, USE”imm”** → Signed, Unsigned Extension of “imm”  
**’LSB, ’MSB** → Least, Most Significant Byte of multiplication  
**’Quo, ’Rem** → Quotient, Remainder of division  
**flags** → Greater and Less than flags  
**AS $\pm$ , LS $\pm$**  → Arithmetic, Logical Shift by + or – shift amount  
**Cmp(, )** → Compare  
**1sComp(, ), 2sComp(, )** → One’s, Two’s complement

### 2-3 Register Summary

SAYAC processor has 16 general purpose registers in a register file for data operations, 16 special purpose registers in a register bank and several single special purpose registers as control status registers (CSRs). TABLE III, shows a summary of the processor registers and their descriptions.  $R_f$  refers to register file and  $R_b$  refers to register bank.

#### NOTE

For more information on CSR , see [Chapter 4](#) section of Special Purpose Registers

**TABLE III: SAYAC Register Summary**

Register Name	Description
$R_f(0)$	Hardwired to zero
$R_f(1)$ - $R_f(14)$	General purpose registers for data operations
CFR- $R_f(15)$	Consists of eight shadow bits, four flags and four status bits.
MAP- $R_b(1)$	Contains the <b>Memory Access Policy</b> for memory protection
MRS - $R_b(2)$	Contains the <b>Memory Region Size</b> for memory protection
TSA - $R_b(3)$	<b>Top Stack Address</b>
IHBA - $R_b(5)$	<b>Interrupt Handler Base Address</b> for interrupt handling
EBA - $R_b(6)$	<b>Exception Base Address</b>
EOA- $R_b(7)$	<b>Exception Offset Address</b>
RSB	<b>Region Size Bank</b>
SRB	<b>Summary Register Bank</b>

## 2-4 Exceptions

SAYAC Processor raises exceptions based on a fault exception mechanism. If a fault is detected the corresponding fault exception handler will be executed. Fault exceptions trap illegal memory accesses and illegal program behavior. The following conditions raises exceptions:

- **Illegal Instruction:** The exception occurs when the program tries to execute any illegal instruction.
- **Memory management faults:** Detects memory access violations to regions that are defined in the Memory Protection Unit (MPU); for example, code execution from a memory region with read/write access only.
  - **Instruction access fault:** The exception occurs when the program tries to access an instruction on an invalid memory location.
  - **Load access fault:** The exception occurs when the programs attempt to do a load an operation on an invalid memory location. For example, trying to load from address which is more than the bound of memory or inaccessible by memory.
  - **Store access fault:** The exception occurs when the programs attempt to do a store an operation on an invalid memory location. For example, trying to store to an address which is more than the bound of memory or inaccessible by memory.
- **Environment Call:** This exception occurs when the programs execute a system call. The system call is realized in SAYAC using MEC instruction.
- **Divide by zero:** This exception occurs when divide by zero happens.

When one of the above exceptions occurs, the controller enters the exception handling states. The datapath units based on exception priority generate the start address of the exception handler through the exception CSRs in the register bank.



For more information on exception handling, see [chapter 5](#) on Exceptions and Interrupts.

## 2-5 Interrupts

SAYAC supports external interrupts for both machine and user modes. External interrupts are handled by a Programmable Interrupt Controller (PIC). The PIC handles interrupt requests coming from different external sources. Prioritization of the interrupts will be handled inside the PIC and the core controller enters the interrupt processing states.



NOTE

---

For more information on interrupt handling, see [chapter 5](#) on Exceptions and Interrupts.

### 3. Memory Model

This chapter describes the SAYAC processor memory model, memory map and memory components.

#### 3-1 Memory Map

SAYAC has an address space of 64 KB for all memory accesses. A word of memory is defined as 16 bits (2 bytes). FIGURE 3.1 shows the system memory map that includes:

- Boot ROM : 2K ROM with base address of 0x0000.
- Instruction (Code) memory: 26K RAM with the base address of 0x0800.
- General data (Data) memory: 28K RAM with the base address of 0x7000. Stack memory is included inside the data memory region.

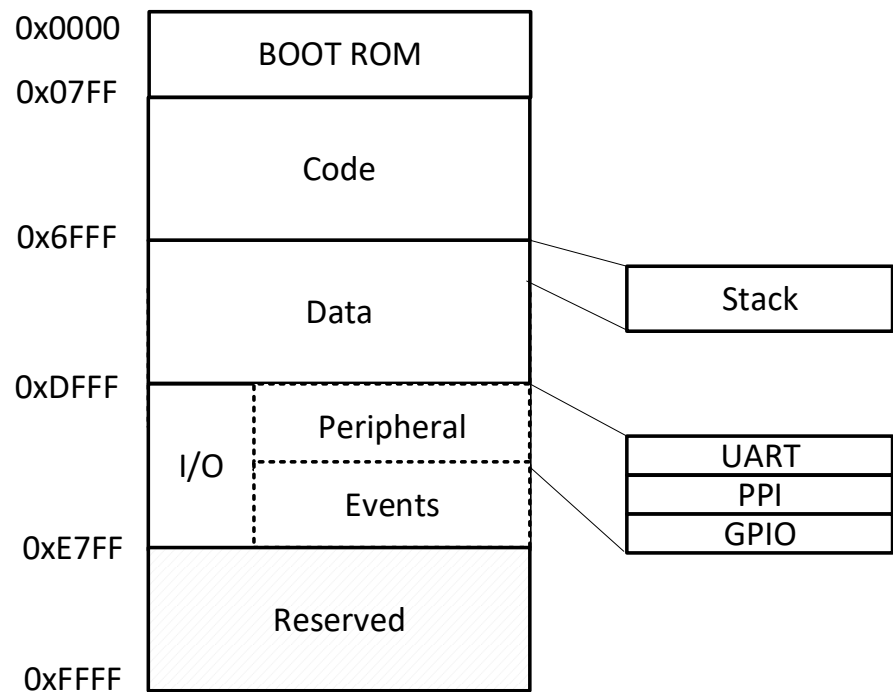


FIGURE 3.1 Memory map of the SAYAC processor

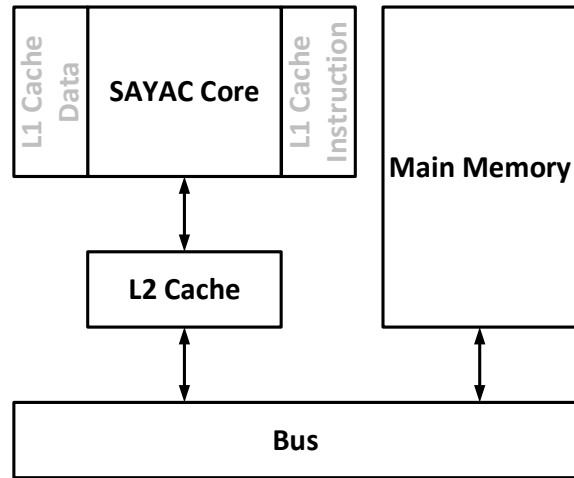


FIGURE 3.2 Memory hierarchy

- I/O device memory: 4K RAM with the base address of 0xE800. This region includes two separate subregions for addressing peripherals and the interrupt events from PIC.
- The rest of memory space is reserved and unused.

### 3-2 Caches

SAYAC processor has a 8KB Cache. At the time of writing, this is a level 1 (L1) unified memory that can hold both data and instructions. In the next versions, this last level cache (L1) can be divided into instruction and data caches and be connected directly to the core logic and handle instruction fetches, and load and store instructions. To improve the memory hierarchy, upper level caches like L2 and L3 can be embedded in the next version. FIGURE 3.2 shows the memory hierarchy of the system. To implement this hierarchy the ports of the cache unit is implemented as shown in TABLE I.

TABLE I: SAYAC Cache Ports

Port Name	Bit		Description
c_data_in	16	input	CPU data that is to be written in cache when c_wr is active
m_blockin	64	input	Block data from the bus for reading from memory
c_address	16	input	CPU addressbus
c_rd/c_wr	1	input	Read and write signals from CPU to cache
m_ready	1	input	Ready signal from memory
c_ready	1	output	Ready signal to CPU
c_dataout	16	output	Cache data that is to be read by CPU when c_rd is active
m_address	16	output	address for addressing memory
m_blockout	64	output	Block data to bus for writing to memory
m_rd/m_wr	1	output	Read and write signals from cache for memory access

## Cache Architecture

The cache for SAYAC is implemented using a set associative cache. This significantly reduces the likelihood of the cache thrashing seen with direct mapped caches, improving program execution speed, and giving more deterministic execution.

Set associative caches are divided into a number of equal sized pieces, called ways. Commonly there are 2-ways or 4-ways. SAYAC cache uses 4-ways. A line refers to the smallest loadable unit of a cache, a block of contiguous words from main memory. The index is the part of a memory address that determines in which line of the cache the address can be found. A way is a subdivision of a cache, each way being of equal size and indexed in the same fashion. The line associated with a particular index value from each cache way grouped together forms a set. The tag is the part of a memory address stored within the cache that identifies the main memory address associated with a line of data.

FIGURE 3.4 shows the SAYAC cache address. The index field of the address is used to select a particular line in each way which in SAYAC is 8bits length.

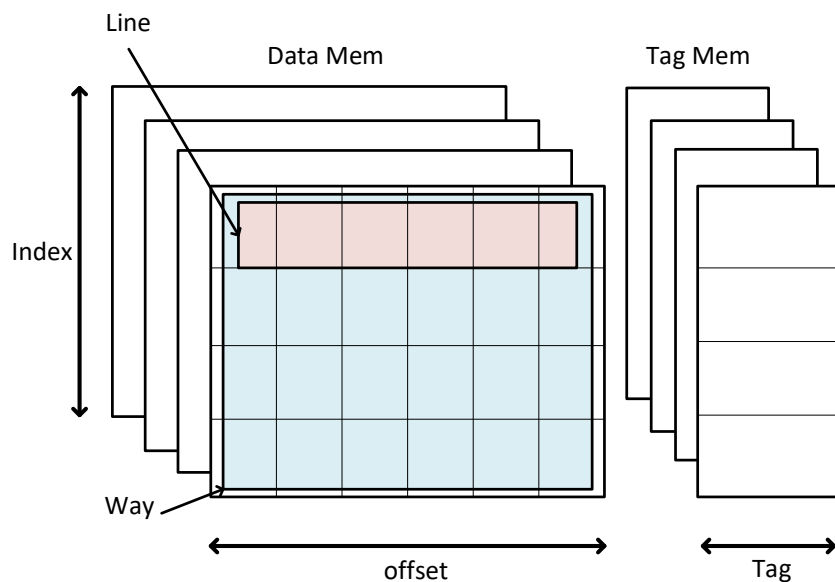


FIGURE 3.3 Cache Terminology

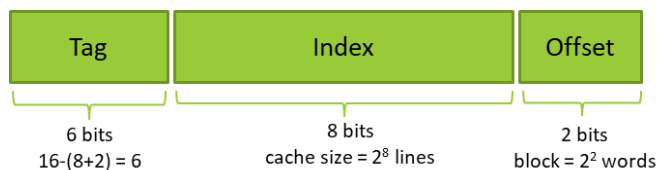


FIGURE 3.4 Cache Address

[illegible]

FIGURE 3.5 One-Way Implementation

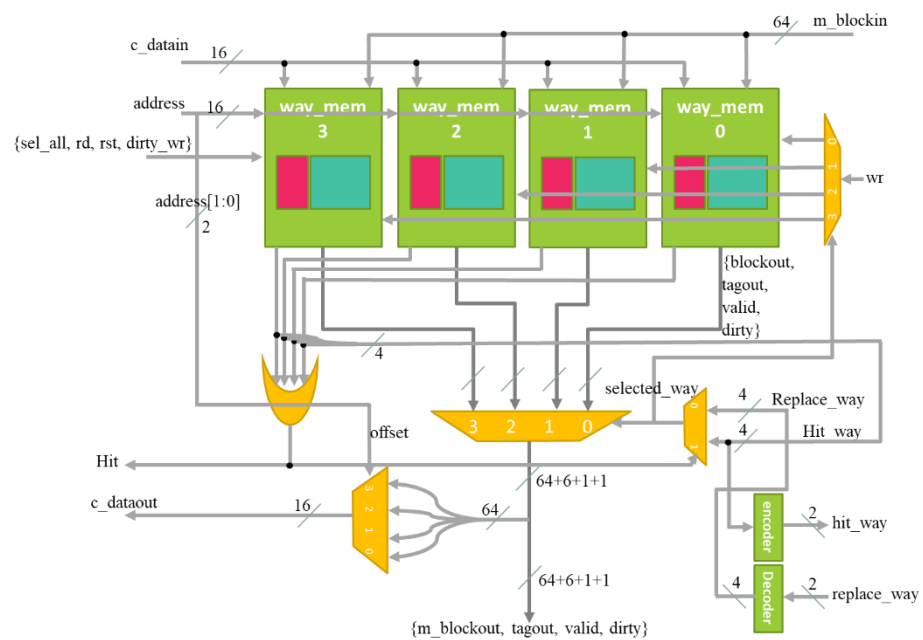


FIGURE 3.6 4-Way Implementation



When the cache controller receives a request from the core it must check to see whether the requested address is in the cache. It does this by comparing a subset of the address bits of the request with tag values associated with lines in the cache. If there is a match and the line is marked *valid* then the read or write will happen using the cache memory (*Cache hit*). When the core requests instructions or data from a particular address, but there is no match with the cache tags, or the tag is not valid, a cache *miss* results and the request must be passed to the next level of the memory hierarchy.

## **Cache Policy**

### **➤ Replacement Policy**

Because of the limitation in cache size compared to memory (or main memory), when it reaches its maximum capacity, when there is a cache miss, the cache controller must select one of the cache lines in the set to be replaced or evicted. The replacement policy is what controls the eviction selection process. The index bits of the address are used to select the set of cache lines, and the replacement policy selects the specific cache line from that set that is to be replaced.

SAYAC cache uses Pseudo LRU (Least Recently Used) replacement policy in which the most recent unused item in the cache will be evicted when a new item needs to be inserted. Also in this version the FIFO replacement policy was implemented and both can be used.

### **➤ Write Policy**

SAYAC cache uses write-back policy which prevents writing to the main memory in the case of hits, reducing memory access time. The cache line holds the most recent data and the main memory holds old data that has not updated. The dirty bit shows the status of the cache line. When the cache controller in write-back writes a value to cache memory, it sets the dirty bit true. If the core accesses the cache line later, it knows by the state of the dirty bit that the cache line contains data not in main memory. If the cache controller evicts a dirty cache line, it is automatically written out to main memory.

## **3-3 Memory Protection Unit**

The Memory Protection Unit controls the access to different regions of the memory map. Each region have a set of attributes and the SAYAC core holds these attributes inside the Register Bank. To implement a protected system, the OS must define a number of regions to cover the different areas in the main memory map. This is done as a static (fixed) scheme, during the boot sequence, that persists while the system is running.

## **Memory Regions**

The memory is partitioned to maximum five regions based on the memory map of FIGURE 3.1:

- Boot
- Instruction
- General Data
- Peripheral

- Events

Each region has the following attributes:

- Region Base address
- Region Size
- Access Policy that includes read, write and execution

The memory is contiguous, so the base address of each region comes at the end of the previous one. Considering that the Boot is the first region, the base address of all regions can be calculated easily using the region size. Region size and access policy are the attributes that will be stored by the OS inside the SAYAC CSR Register Bank.

### ➤ Region Size

The region size is specified with maximum 5 bits. This 5-bits show the maximum number of pages that each region can cover. Each page is considered as 1K. MPU region size register is the second register of the register bank named **MRS**. FIGURE 3.7 shows each region size bit-width in **MRS** register.

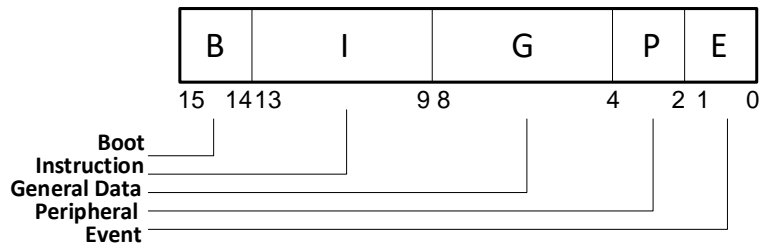


FIGURE 3.7 *MRS* register configuration

### ➤ Access Policy

The access policy for each region will be stored inside the third register of the register bank named **MAP**. The R, W, and X bits, when set, indicate that the MPU entry permits read, write, and instruction execution, respectively. TABLE II shows the typical access permissions for these 5 regions.

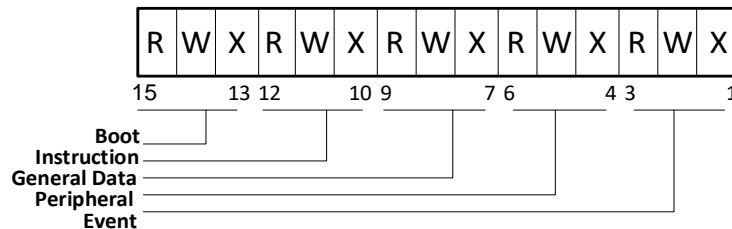


FIGURE 3.8 *MAP* register configuration

**TABLE II:** Region access permissions

Region	Policy		
Boot	R	--	X
Instruction	R	--	X
General Data	R	W	X
Peripheral	R	W	--
Event	R	W	X

Two special registers named RSB and SRB will make carbon copies of the MRS and MAP registers. These values will be used for region address calculations. The information will be used for generating the region address boundary. This is done inside the core using a hardware **Address Region Check** unit. Based on the access policy of the specified region and the calculated boundaries, the address bus will be checked. The outputs are signals that represent memory management faults including instruction, load and store access faults.

 NOTE

For more information on ARC unit, see [Chapter 4](#) on SAYAC core architecture.

## 4. SAYAC Core Architecture

This chapter describes the SAYAC processor core architecture, including the datapath and controller units.

### 4-1 Overview

FIGURE 4.1 shows a block diagram of the SAYAC core. The core includes the following general units:

- Generic datapath registers
- Special purpose registers
- Register file
- Immediate unit
- Address unit
- Execution unit
- Exception unit
- Memory protection unit

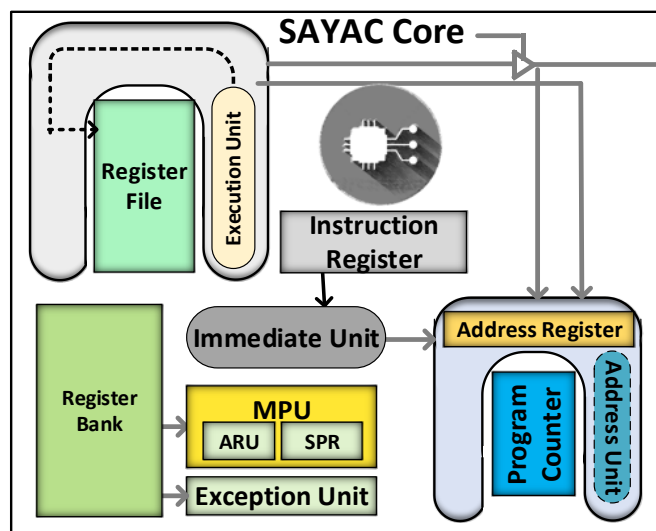


FIGURE 4.1 SAYAC core block diagram

## 4-2 Datapath

FIGURE 4.2 shows the datapath of the SAYAC processor. Details of the datapath components will be described in the following sections:

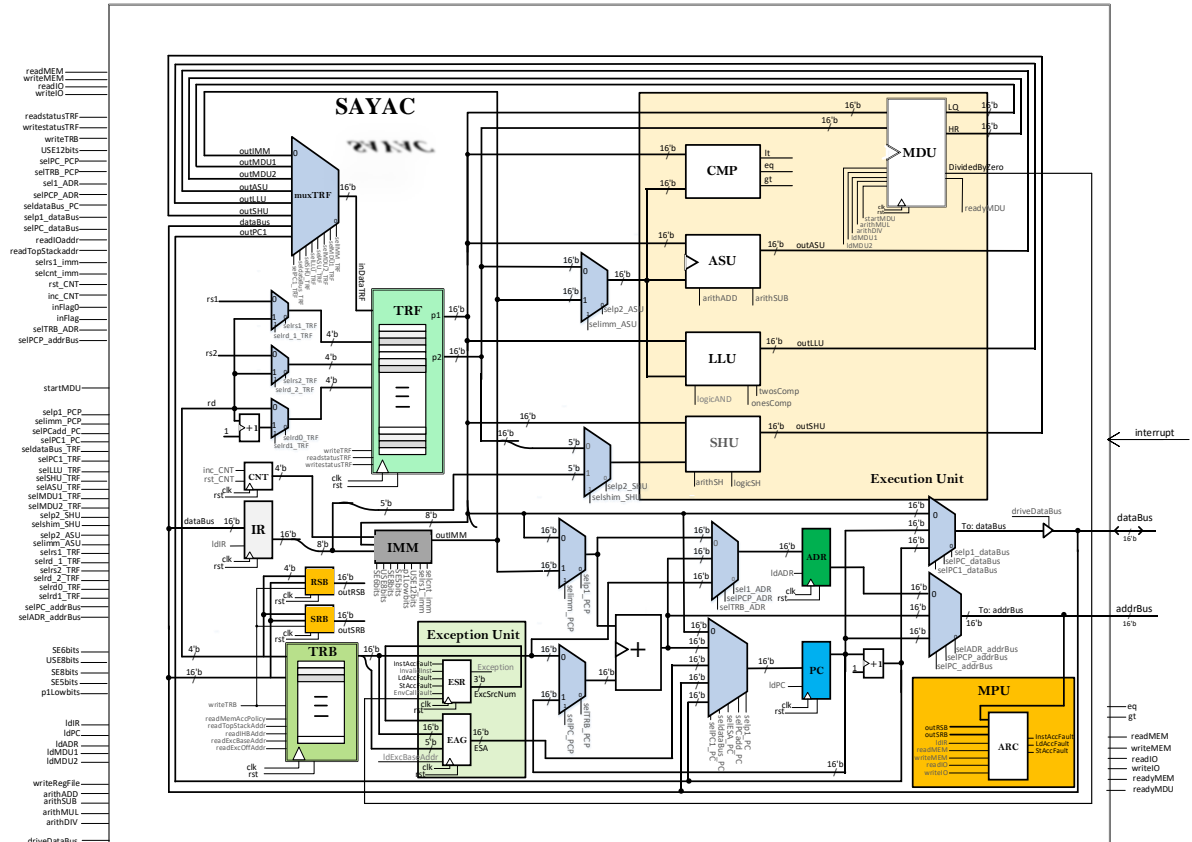


FIGURE 4.1 SAYAC core datapath

### ➤ Generic datapath registers

Generic registers inside the core are 16-bit registers with asynchronous reset and a load signal for updating register contents.

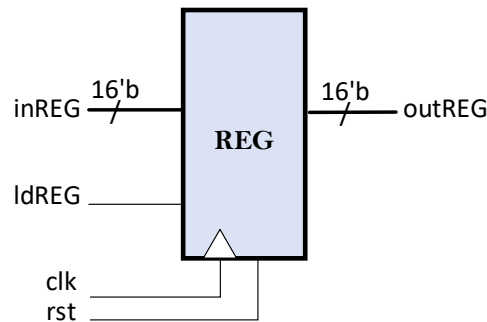


FIGURE 4.2 SAYAC core block diagram

Excluding the 32 general-purpose registers of the Register File, SAYAC has 3 additional registers inside the datapath:

- Program Counter (PC)
- Instruction Register (IR)
- Address Register (ADR)

#### ➤ Special Purpose Registers

SPR of the SAYAC core are implemented by a register bank and also one control and flag register in the register file i.e., R<sub>f</sub>(15).

#### • The Register Bank, TRB:

The register bank includes the control status registers for memory protection unit and exception handling. Rb(1) (MAP) and Rb(2) (MRS) are two CSRs as the memory access policy and memory region size for generating the region boundaries in MPU. Rb(3) (TSA) holds the address of the top of stack.

Rb(5) (IHBA) holds the Interrupt handling base address. This is used in interrupt handling states for executing the interrupt service routine. Two other CSRs are Rb(6) (EBA) and Rb(7) (EOA) that hold the exception base and offset addresses, respectively. These register values are used in the exception unit for generating the exception service address.

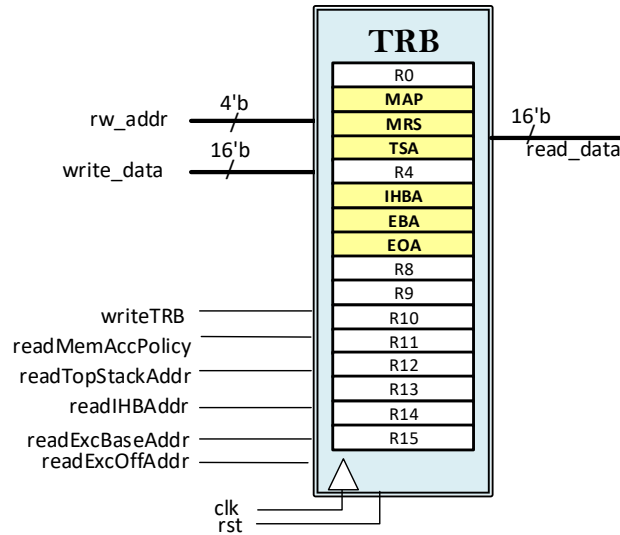
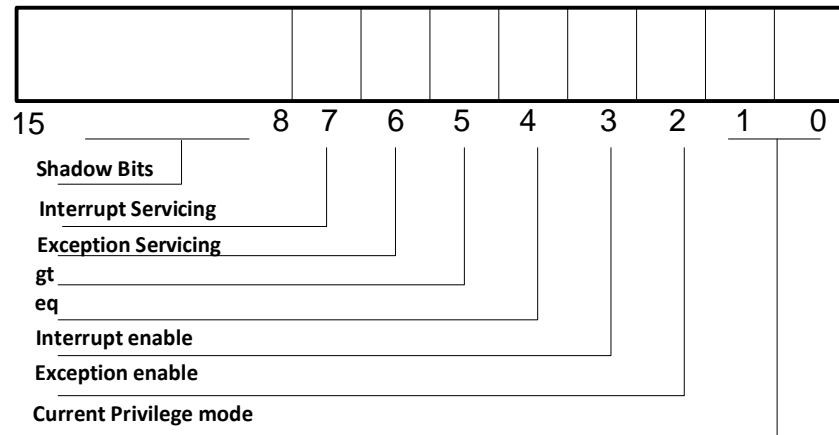


FIGURE 4.3 The register bank of the SAYAC core



- **Control and Flag Register, R<sub>f</sub>(15):**

The register in the location 15 of the register file includes the control and shadow flags. FIGURE 4.4 shows the configuration of this register. The most significant 8 bits are used for shadow instructions. When the corresponding shadow bit of the shadowed instruction is set, the shadow mode will be executed. The least 8-bits of the CFR register hold the control flags.

FIGURE 4.4 Control Flag Register bit configuration

- ❖ CFR [1:0] are used for storing the current privilege mode. Based on the privilege mode (machine or user) any attempt to perform operations not permitted by the current privilege mode cause an exception to be raised. For machine and user mode, it is set to 00 or 01 respectively.
- ❖ CFR [3:2] are the global interrupt and exception enables of the current privilege mode.
- ❖ CFR [5:4] are *gt* and *eq* flags. These flags will be set as the result of compare instructions and will be accessed during Jump instructions.
- ❖ CFR [7:6] are the interrupt and exception servicing flags. When an interrupt or exception is being handled, the corresponding bits in the CFR register will be set to 1 ; so that the interrupt or exception control unit can release the event signal to prevent redundant reserving.

➤ **The Register file, TRF**

SAYAC register-file is a multi-port structure with two 4-bit read addresses and their 16-bit output ports. The two read operations can be done simultaneously

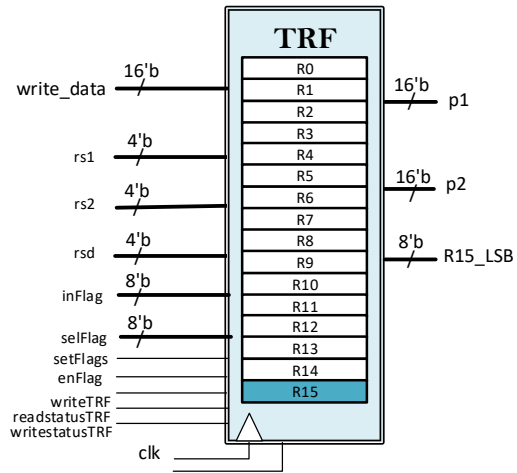


FIGURE 4.5 The register file of the SAYAC core

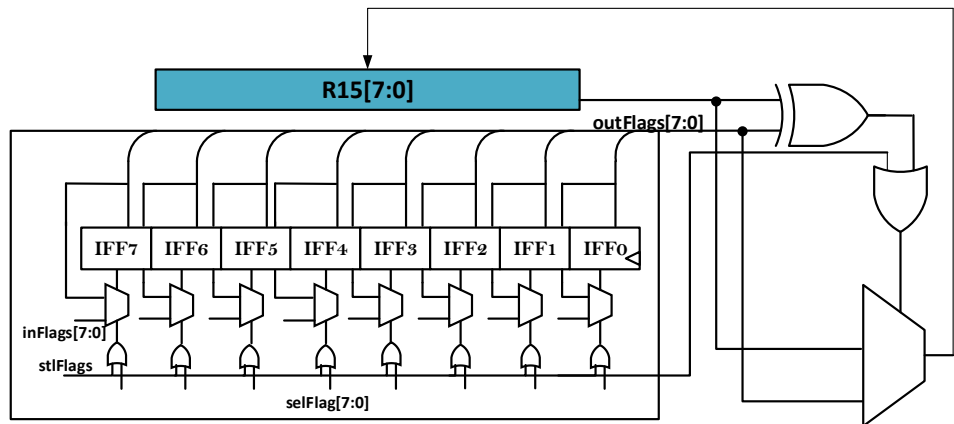


FIGURE 4.6 Logic for writing to  $R_i(15)$

In addition, the register-file has a 16-bit write port and its corresponding 4-bit write address port. Two reads and a write can be active simultaneously. However, the write operation will only take place on the active edge of the clock during which write is enabled. FIGURE 4.5 shows the block diagram of the register file of the SAYAC core.

As mentioned before, the eight least significant bits of the  $R_i(15)$  are control flags that can be accessed both for reading and writing. FIGURE 4.6 shows the logic for writing control flags into  $R_i(15)$ . To do this, eight flip flops (IFF) are used to set the corresponding flags.



### ➤ Immediate Unit, IMM

Since the I-type instructions in SAYAC ISA can support different immediate length, IMM unit generates the signed and unsigned extended version of the immediate with 5, 6 and 8 bits based on the instruction type given encoded in the instruction register (IR). Also, this unit will generate the unsigned extension of the counter value for interrupt processing handling. FIGURE 4.7 shows the port configuration of the immediate unit.

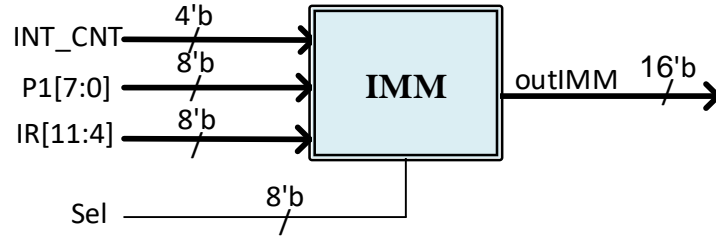


FIGURE 4.7 Immediate Unit

The immediate output is set according to the following configuration of the signal sel:

**TABLE I:** *sel* signal configuration

Instruction	Opcode	sel
I-Type	001011	0x40
	0100	0x20
	0101	0x10
	0110	0x08
	1011	0x10
	1100	0x10
	111100	0x05
	111101	0x05
Interrupt	---	0x01

### ➤ Execution Unit

The execution unit is the main calculation unit that performs arithmetic and logic operations. This unit includes the following modules:

- **Compare Unit, CMP**

The compare unit is a combinational circuit that takes two 16-bit input operands and outputs the result of two comparisons: greater-than (gt) and equal-to (eq). The signal *signCMP* allows to choose if the comparison is performed on signed or unsigned numbers. FIGURE 4.8 shows the interfacing ports.

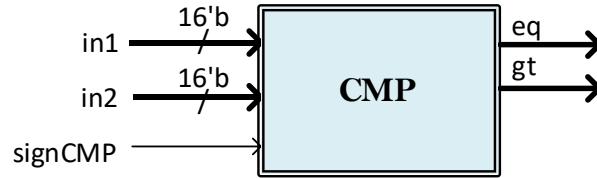


FIGURE 4.8 Compare Unit

- **Add Subtract Unit, ASU**

Based on the control signals *arithADD* and *arithSUB*, the adder and subtractor unit will perform addition or subtraction. Two 16-bit inputs are *in1* and *in2* and the result is 16-bit *outASU*. For implementing the adder a Carry Lookahead Adder is used.

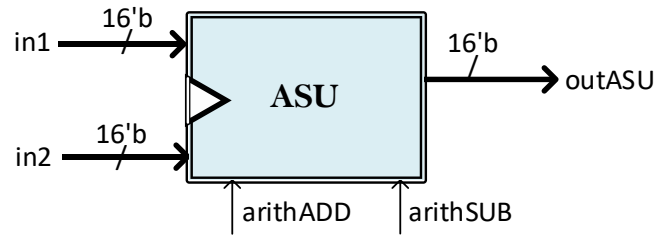


FIGURE 4.9 Add-Subtract Unit

- **Logical-Logic Unit, LLU**

The Logical Logic Unit performs two logic operations. AND and Complement. Based on the control signal *twosComp* being set to one or zero, Two's or one's complement will be executed respectively. The logical AND of two input sources *in1* and *in2* will be executed when the *logicAND* is set to one. FIGURE 4.10 shows the port configuration of the LLU unit.

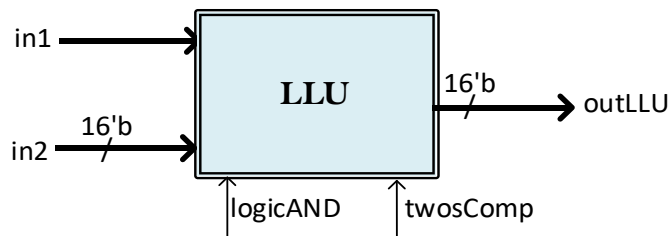


FIGURE 4.10 Logical Logic Unit

- **SHift Unit, SHU**

The shift unit performs two kinds of shift operations: arithmetic, and logical shift. Two control signals from the core controller *arithSH* and *logicSH* indicate the shift type. As shown in FIGURE 4.11 one 16-bit input source as the input for being shifted, *in1*, and one 5-bit input as the length of shift operation, *in2*, will be fed to the module. The

direction of the shift operation is determined based on the sign of the *in2* source signal. If *in2* is a negative value, a left shift operation will be executed in either of arithmetic or logical shift. Vice versa, if *in2* is a positive value a right shift operation will be executed. Note that the sign of *in2* is determined by its most significant bit and the length of shift will be  $in2[3:0]$ .

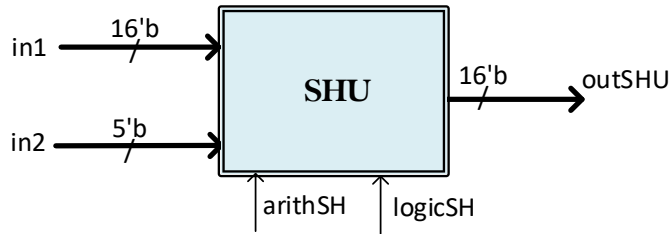


FIGURE 4.11 SHift Unit

- **Multiply- Division Unit, MDU**

The MDU unit performs signed and unsigned multiplication and division on two 16-bit operands *in1* and *in2*. MDU start to work when the control signal *startMDU* is set to one. Two control signals *arithDIV* and *arithMUL* specify the type of operation. When the operation is completed, *readyMDU* will be set to one. If the multiplication or division is to be signed, *signMDU* will set to one, the operation will be performed by the positive values using the two's complement of the operands and when completed output will be complemented for the negative results. Note that signed multiplication and division are enabled when using the shadow modes of MUL and DIV instructions.

As shown in FIGURE 4.12, the outputs of the MDU module are Low/Quotient (LQ) and High/Remainder (HR). The result of multiplying two 16-bit operands is a 32-bit value. This 32-bit will be represented in two low and high halves. The HR and LQ are the most and least significant 16-bits of the multiplication result. These two values will be stored in  $R_r(r_d)$  and  $R_r(r_d+1)$  when MUL instruction is executed. For division of two 16-bit operands HR and LQ show the remainder and quotient respectively. These two values will also be stored in  $R_r(r_d)$  and  $R_r(r_d+1)$  when DIV instruction is executed.

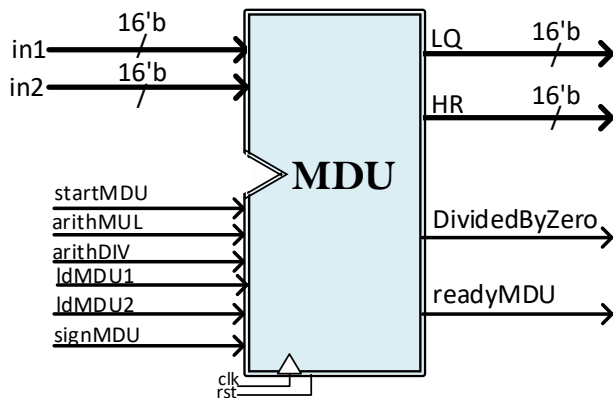


FIGURE 4.12 MDU Unit

When the divisor in DIV instruction is zero, the signal *DividedByZero* will be set to one and this will raise an exception.

For the ASIC implementation of the SAYAC core, three version of Radix-2, Radix-4 and Radix-16 is implemented. In the following Radix-4 is described below to show the general procedure.

#### ❖ Radix-4 Multiplier

A 16-bit sequential Radix-4 signed and unsigned multiplier that includes a separate datapath and controller. The circuit implements the following algorithm. Starting from the least significant bit and by adding a zero on the right side of the multiplier, every 3-bits block will be checked, and a partial product will be selected based on the encoding shown in TABLE II. Based on the sign of the partial product, zero, multiplicand or 2\*multiplicand will be added or subtracted to the previous partial sum product.

**TABLE II:** Radix-4 encoding for partial product

Block	Partial Product
000	0
001	1*multiplicand
010	1*multiplicand
011	2*multiplicand
100	-2*multiplicand
101	-1*multiplicand
110	-1*multiplicand
111	0

Based on this algorithm, the datapath of radix-4 is implemented as shown in FIGURE 4.13. It includes one register for storing the multiplicand, one shift register for storing the multiplier and shifting the blocks, one register for storing the partial sum product, a multiplexer for selecting the encoding scheme and an ADD/SUB unit.

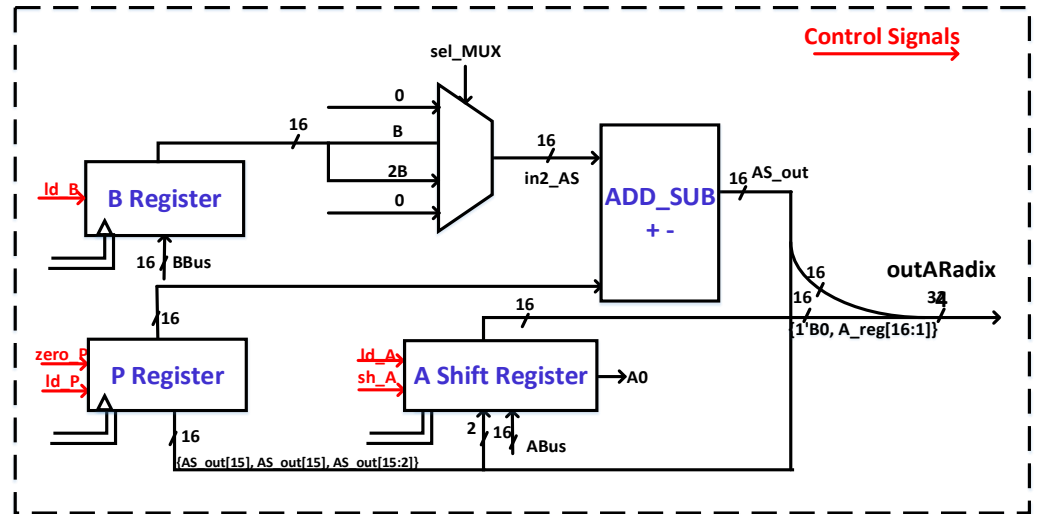


FIGURE 4.13 Radix-4 Multiplier datapath

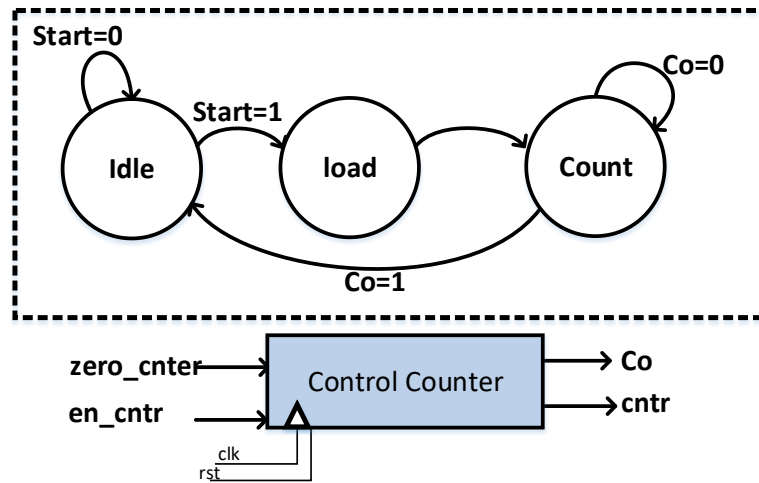


FIGURE 4.14 Radix-4 Multiplier controller

The control signals are generated in a controller as shown in FIGURE 4.14. A counter is instantiated for counting for the half of the multiplier bit width. Three states are defined in the finite state machine: Idle, load, and Count. Idle state is waiting for *Start* signal. Load is the initialization state in which A and B registers are loaded. P register is set to zero and the counter is reset. When *start* is set to 0, the state is set to Count. In this state, the counter is incremented and control signals for the shift are set, the arithmetic shift right is performed until the carry out is not zero.

#### ❖ Radix-2 Divider

The SAYAC divider uses the Radix-2 division algorithm. FIGURE 4.15 shows the datapath of the core divider.

- Registers initialization to their corresponding values. The dividend is assigned to Q, the divisor to M, and result to 0.
- An additional counter set to the number of bits (n) is used to count the algorithm iterations.
- The content of register R and Q is shifted left as if they are a single unit.
- The content of register M is subtracted from R and result is stored in sub.
- The most significant bit of the sub is checked: if it is 0, the least significant bit of Q is set to 1; otherwise, if it is 1, the least significant bit of Q is set to 0 and the value of register R is restored (i.e., the value of sub before the subtraction with M).
- The value of counter n is decremented.
- If the value of n becomes 0, the loop ends, otherwise it repeats from step 2.

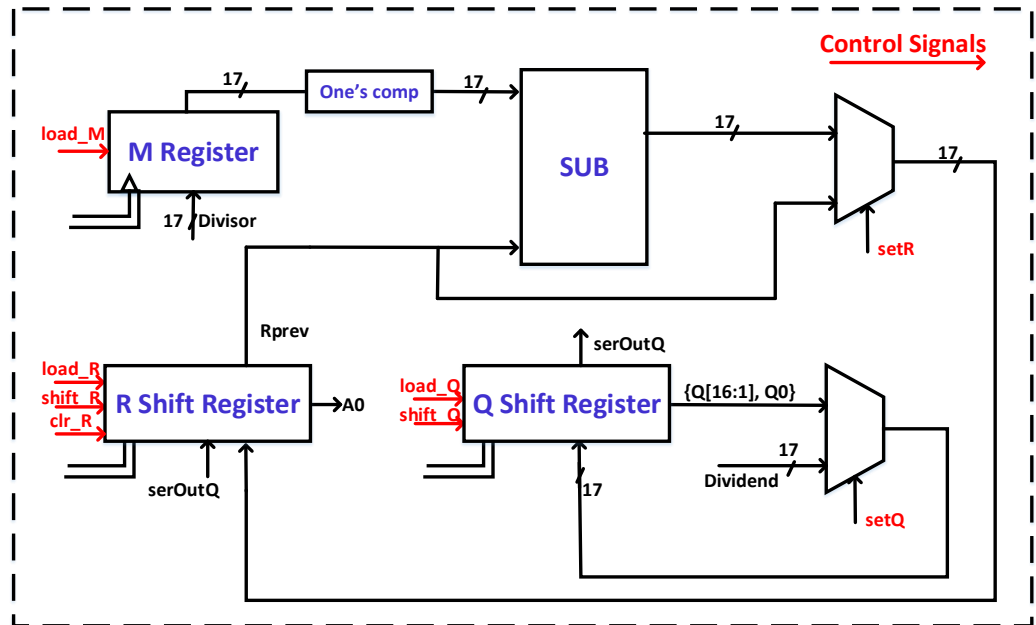


FIGURE 4.15 Radix-2 divider datapath

FIGURE 4.16 shows the controller of the radix-2 divider. The finite state machine has three states; *Load*, *Shift* and *Sub*. In the first *Load* state, while waiting for the division *Start* signal, the values of R, Q and M register will be initialized. When *Start* is issued shift and subtraction will be performed in two consecutive states *Shift* and *Sub*. The counter will be decremented each time and the division will be completed by returning back to the *Load* state when the counter reaches to zero. Otherwise the shift and sub will be repeated.

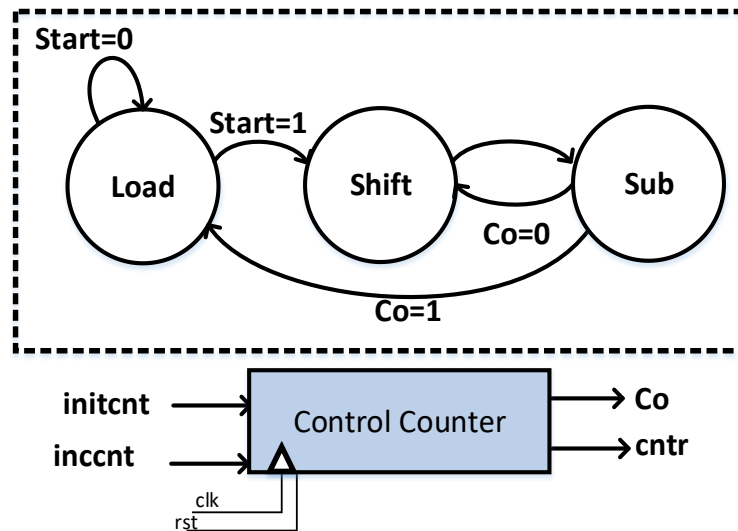


FIGURE 4.16 Radix-2 divider controller

### ➤ Exception Unit

As mentioned in Section 2-4, there are six exception causes in SAYAC Core and each cause has a specific exception handler. Modules in the exception unit, are responsible for generating the start address of exception handler process. This address will be calculated based on the following relation using the exception base address and offset that has been stored in the *EBA* and *EOA* registers respectively.

$$\text{Exception handler address} = \text{Base Address} + 2^{\text{ExcSource}} * \text{offset} \quad (1)$$

The hardware implementation of the above relation is described below:

- **Exception Source Register, ESR**

The ESR unit detects if an exception occurred and outputs the exception source number. For this purpose, a given ID number is dedicated to each exception cause. FIGURE 4.17 shows the port configuration of this unit. The inputs are the fault flags generated from the controller. Based on the encoding of the TABLE III, *ExcSrcNum* will be generated. Also an Exception flag shows the exception occurrence.

**TABLE III:** Exception Source Numbers

Exception Cause	Exception Source Number
InsAccFault	1
InvalidInst	2
LdAccFault	3
StAccFault	4
EnvCallFault	5
DividedByZero	6

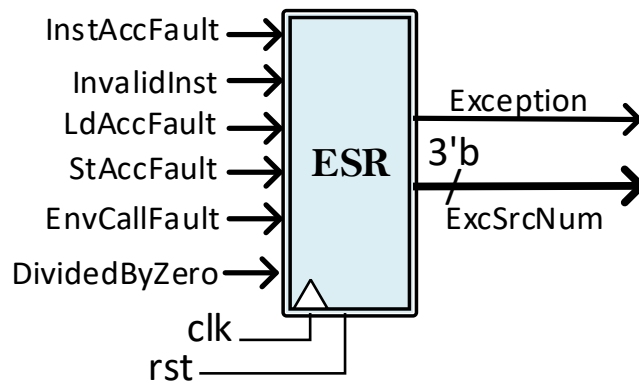


FIGURE 4.17 ESR unit

- **Exception Address Generation, EAG**

This unit implements the exception handler base address based on the equation (1). The inputs as shown in FIGURE 4.18, are the exception source number, the base address

and the offset. The offset value will be shifted for the number of exception source and will be added to the exception base address. The result would be the value of address bus after exception detection.

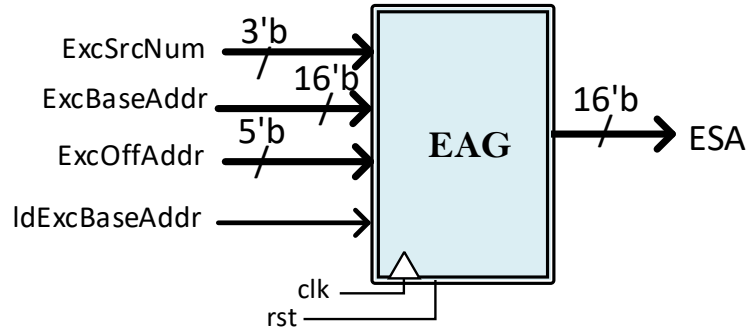


FIGURE 4.18 EAG unit

### ➤ Memory Protection Unit

As mentioned before, MPU will detect memory management faults including instruction, load and store access faults. This is done by checking if the current address bus is within the specified boundaries of the memory regions. This is done by ARC unit explained below:

#### • Address Region Check, ARC

ARC is responsible for two main operations. First it estimates the memory region boundaries based on the region sizes and second it checks the address region for access faults. Two inputs *outRSB* and *outSRB* are the outputs of two special registers RSB and SRB. These registers hold the carbon copies of MAP and MRS registers of the register bank.

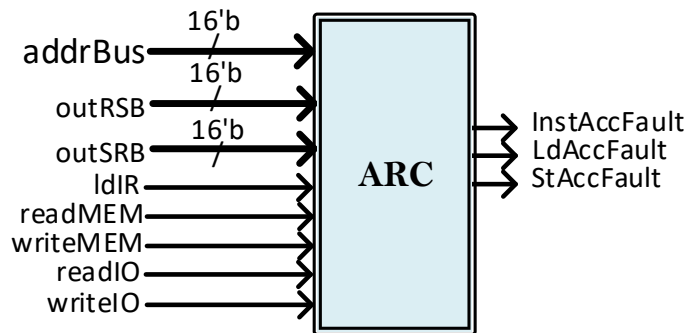


FIGURE 4.19 ARC unit

For checking the address region, control signals indicating the memory access *readMEM*, *writeMEM*, *readIO*, *writeIO* and instruction access *ldIR* are fed to the ARC unit. First the address region is determined and then for each case of accesses the access policy will be checked. If the wrong policy is chosen the ARC unit issues the corresponding access fault.



### 4-3 Controller

The control unit generates the control signals for executing instructions. SAYAC controller works based on five main states as shown in FIGURE 4.20. The first state is “*Fetch*”, in which the instruction will be fetched and based on its opcode, a decision will be made on the next state of execution.

Before executing the instruction, the control unit checks if any interrupt or exception has occurred. Three interrupt processing states, “*ISP1*”, “*ISP2*” and “*ISP3*” and two exception processing states, “*EPS1*” and “*EPS2*” are dedicated to the interrupt and exception handling respectively.

“*Exec1*”, “*Exec2*”, “*Exec3*” and “*Exec4*” are the states for executing instructions. The bubbles around “*Exec1*” show the conditions for all instructions based on which the next state will be selected. Single cycle instructions will go back to the “*Fetch*” state while multiple cycle instructions like *LDR*, *STR*, *MUL* or *DIV* set the next state to “*Exec2*”.

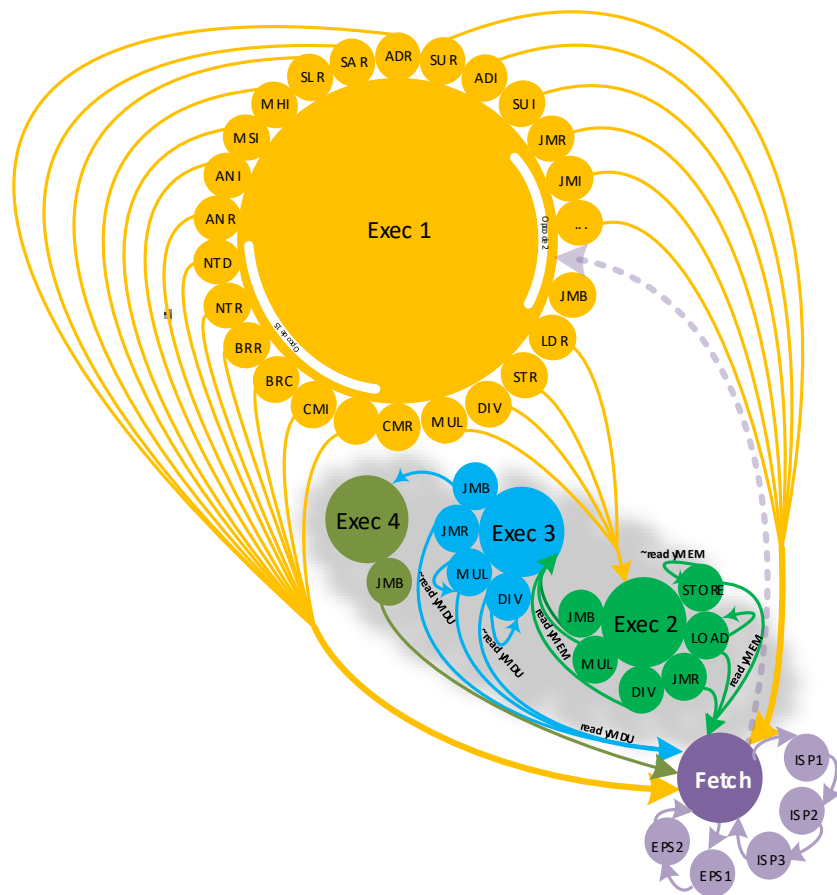


FIGURE 4.20 SAYAC controller unit

For instructions like MUL and DIV that needs to wait for more than 2 cycles to complete the operation, next state will be set to “*Exec3*”.

**TABLE IV:** Description of the controller states

States	Description
Fetch	The initial state of the controller in which all the control signals are reset and the instruction in the current location is fetched.
IPS1	In this state the address unit signals are issued to put the handler address on the addressbus, and the databus will be derived by $R_4(15)$
IPS2	In this state the address unit signals are issued to put the handler address plus one on the addressbus, and the databus will be derived by PC
IPS3	In this state the PC gets the value at location of the handler address plus two
EPS1	In this state, the address bus gets the exception base address and saves the PC
EPS2	In this state, signals for EAG unit are issued for generating the exception handler address and the PC gets this address
Exec1	In this state datapath is configured to execute the instructions. Load, Store, Jump, Arithmetic or Logic operations will be executed based on the opcode [15:8].
Exec2	The datapath is configured to perform 2 or more cycle instructions like Load, Store, Jump, MUL or DIV
Exec3	The datapath will be configured for multi-cycle instructions like MUL and DIV
Exec4	The datapath will be configured for the last cycle of the return (JMB) instruction.

## 5. Exception and Interrupts

This chapter describes the SAYAC processor exception and interrupt handling procedure.

### 5-1 Exception

An exception is any condition that requires the core to halt normal execution and instead execute a dedicated software routine known as an exception handler. Exception that are handled in SAYAC processor as described in Section 2.4 are for handling the memory access faults, illegal instructions, divide by zero and environmental calls.

When an exception occurs, the core saves the current status and the return address. For each exception the exception handler starts from a fixed memory location. This location can be set inside the system register by a privileged software, and they will be executed automatically when respective exceptions are taken. In SAYAC processor only the beginning of the exception vector table will be stored by the software. This is called Exception Base Address (EBA) and all the exception handlers are located at an offset from this base address. The offset value is also configurable by the software program and will be stored at the core register bank. The offset is called Exception Offset Address (EOA).

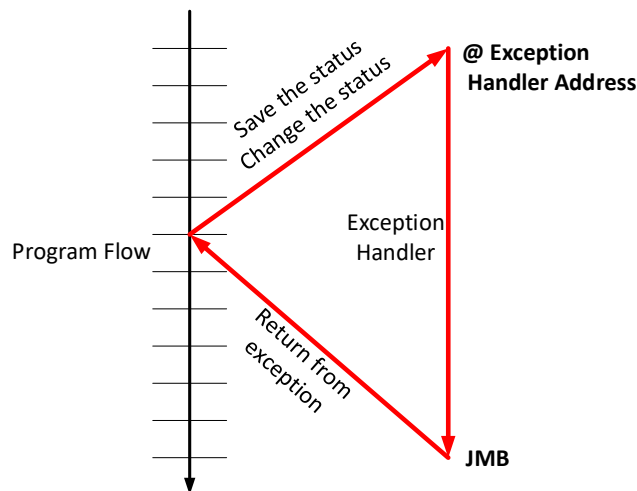


FIGURE 5.1 Exception handling flow

For generating the exception handler address for each exception type, equation (1) will be implemented. To understand the address generation, an example is shown in TABLE I. For this example, the EBA is 0x1200 and EOA is 0x0008.

**TABLE I:** Exception address generation

Exception Cause	Offset	Exception Handling address
InsAccFault	0x0008	0x1208
InvalidInst	0x0010	0x1210
LdAccFault	0x0020	0x1220
StAccFault	0x0040	0x1240
EnvCallFault	0x0080	0x1280
DividedByZero	0x100	0x1300

### • Exception Handling

When an exception occurs, the SAYAC core uses the following exception procedure

- At the fetch state the controller checks the internal exception flag (from ESR Unit) and exception enable flag at  $R_t(15)[2]$ .
- If both are issued, then moves to the first exception processing state.
- In the ESP1 state of the control unit, the exception enable flag will be set to zero ( $R_t(15)[2]=0$ ) and the exception servicing flag to 1 ( $R_t(15)[6]=1$ ).
- The PC will be stored at the location of EBA.
- In the ESP2 state of the control unit, the EOA will be read from the register bank.
- The EAG unit in the datapath unit, generates the corresponding exception handler address
- Set the next PC to the location of exception handler address generated from EAG unit.

### • Exit from an exception handler

- Return instruction JMB is used to restore the return address after handling the exception.
- PC will be set to the return address stored in EBA.

## 5-2 Interrupt

Like all embedded processors, SAYAC will respond to external interrupt events. An external interrupt controller, controls interrupt requests from different external interrupt sources. At the time of writing, no design specific of interrupt controller is developed and any general programmable interrupt controller can be used. The PIC signals an interrupt to the processor core after prioritizing between interrupt sources. It also provides the address in memory for the interrupt service routine.

Once an interrupt is detected before moving to the ISR address, the core stores the current status of control flag register  $R_f(15)$  and the current PC value. Fixed consecutive memory locations are dedicated for saving these contexts. The first address is called Interrupt Handler Base Address (IHBA) and will be stored inside the register bank by the software program.

The third location from the IHBA, stores the address of the interrupt service routine provided by PIC and a jump to this address, starts the interrupt service.

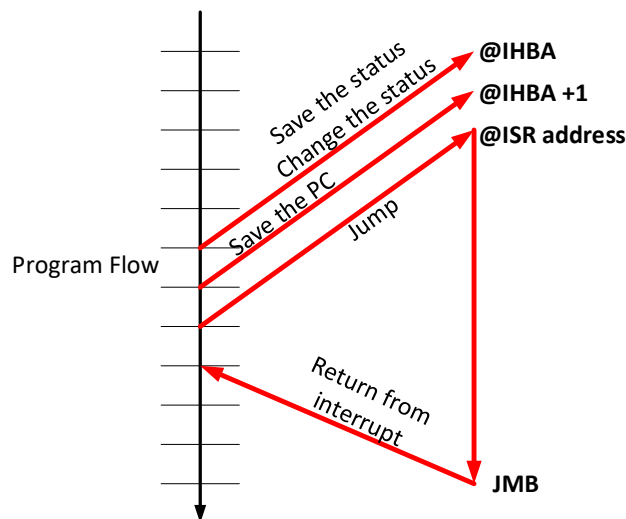


FIGURE 5.2 Interrupt handling flow

- **Interrupt Handling**

The following steps show how SAYAC core handles an interrupt:

- At the Fetch state the controller checks the external interrupt flag (from PIC) and interrupt enable flag at  $R_f(15)[3]$ .
- If both are issued, then the controller moves to the first interrupt processing state.
- In the ISP1 state, the contents of  $R_f(15)$  is stored into the first location of IHBA.

- The interrupt enable flag will be set to zero ( $R_f(15)[3]=0$ ) and interrupt servicing flag to 1 ( $R_f(15)[7]=1$ )
- Increment the counter to increase the offset from the IHBA location.
- In the ISP2 state of the controller unit, the PC will be saved at the second location of IHBA and the counter will be incremented for the next location.
- In the ISP3 state of the controller unit, the next PC will be set to the location of Interrupt service routine.
- For exiting the handler, the PC will be set to the return address by using JMB instruction.

## 6. Interfaces

This chapter describes the SAYAC system interfaces including bus interface and cache interconnects.

### 6-1 System Bus

The bus system at the time of writing supports a shared bus implementation. This bus system supports multiple bus masters. The basic flow of the bus operation is as follows:

- The arbiter determines which master is granted access to the bus.
- When granted, a master initiates transfers on the bus.
- The decoder uses the high order address lines to select a bus slave.
- The slave provides a transfer response back to the bus master and data is transferred between the master and slave.

### 6-2 Bus components

This section describes the bus components and their signaling briefly. The signals used in the bus interface are described below:

**TABLE I:** Bus interface signals description

Signal		Description
clk		clk is the primary clock, which is used to time all bus transfers. Positive edge of the clock is used.
rst		Active high reset signal that is used to reset the bus.
address		The 16-bit address bus provides the address of the transfer. All transfers are memory-mapped. The decoder uses the address bus (usually the higher order bits) to determine which bus slave is to be accessed.
req		The request signal from a master to the arbiter which indicates that the master requires the bus.
gnt		The grant signal from the arbiter to a bus master indicates that the bus master is currently the highest priority master requesting the bus
select		Chipselect signal for slaves that are responsible for supplying a transfer response.
data_in		16-bit data that is to be written.
data_out		16-bit data that will be read.
Control information	read/write	Determines the transfer direction. Read and write are used to show the read and write accesses.
	ready	Transfer must be done when the previous transfers are completed, or the component is not busy. This is shown by ready being HIGH.

- **Bus Slave**

Bus slave responds to transfers initiated by bus masters within the system and is dependent on the master. If a slave is selected for data transfer, read or write operations will be performed on its memory-mapped addressable space under the control of the master. FIGURE 6.1 shows the slave interface diagram.

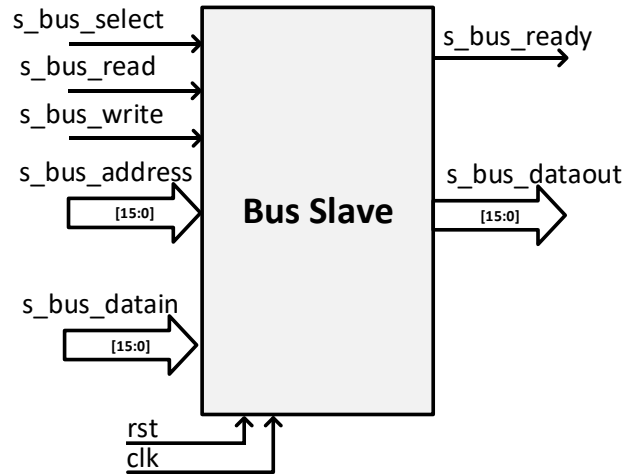


FIGURE 6.1 Bus slave interface

- **Bus Master**

Master is an independent component that can request to perform the read or write operation to any slave in the system. FIGURE 6.2 shows the master interface.

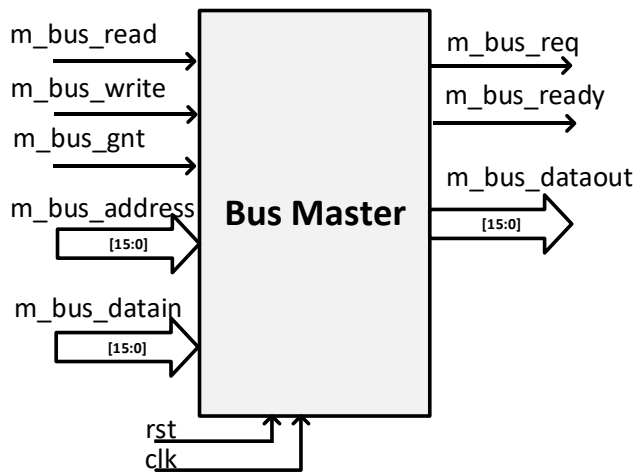


FIGURE 6.2 Bus master interface



- **Bus Arbiter**

The role of the arbiter unit is to arbitrate the bus requests among the masters so that only one master is allowed to access the bus. Every bus master has a two wires, request (*req\_x*) and grant (*gnt\_x*) interface to the arbiter and on every cycle the arbiter uses a prioritization scheme to decide which bus master is currently the highest priority master requesting the bus.

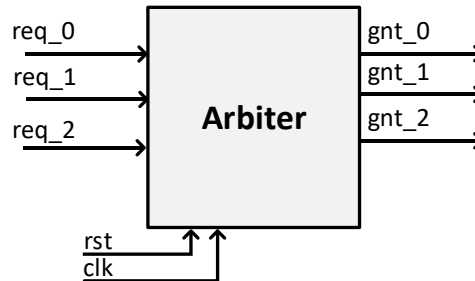


FIGURE 6.3 Bus arbiter

- **Bus Decoder**

The decoder generates a select signal for each slave on the bus system. The decoder greatly simplifies the slave interface and removes the need for the slave to understand the different types of transfer that may occur on the bus.

FIGURE 6.4 shows a typical scheme for a bus with multi master and slave ports.

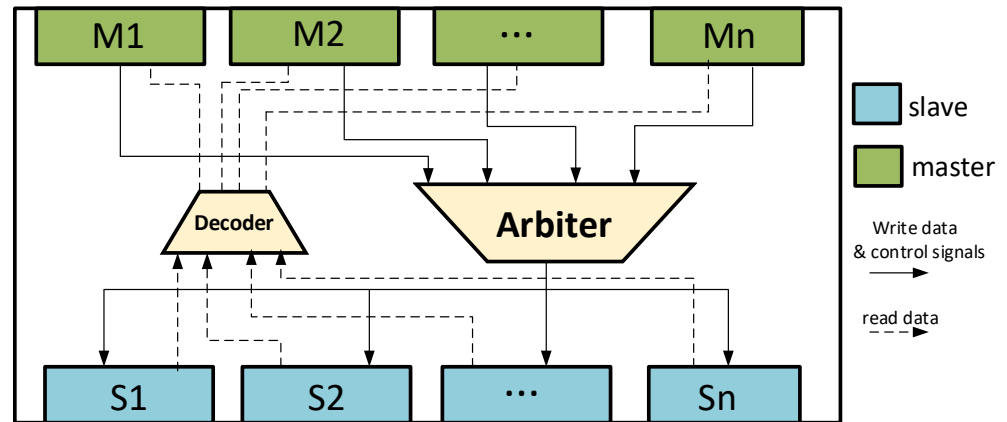


FIGURE 6.4 Bus interconnection scheme

### 6-3 Cache Interface Unit

As shown in the cache unit, in [Chapter 3](#), the data bit width for the cache input and output ports are 64-bit. The bus structure supports data and address width of 16-bit for both master and slave ports. Therefore, the system needs an interface to synchronize the

data transfer between the cache and bus units. FIGURE 6.5 shows the port configuration of the CIU.

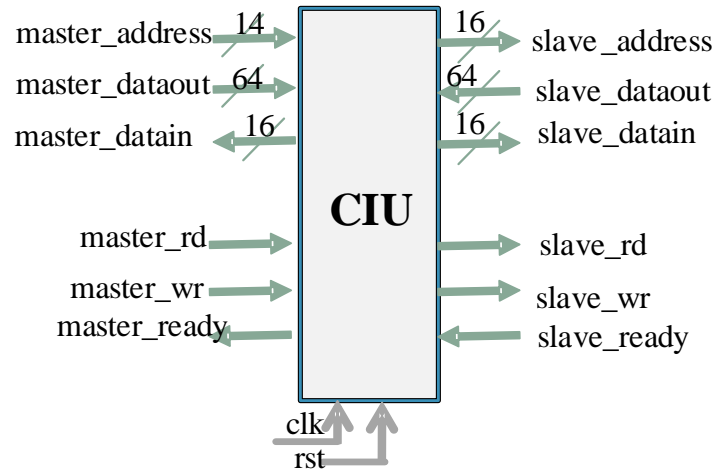


FIGURE 6.5 CIU port configuration

FIGURE 6.6 shows the RTL implementation of the cache interface unit.

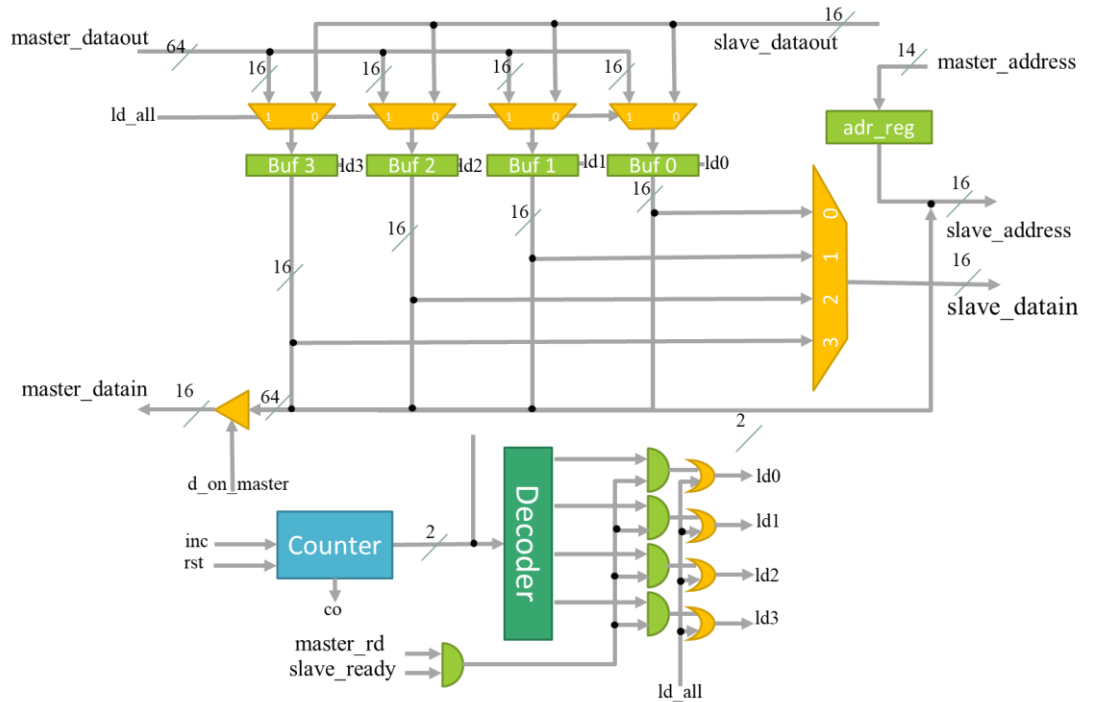


FIGURE 6.5 RTL implementation of CIU

## 7. System User Interface

This chapter describes the SAYAC system user interface and includes the simulation setup and examples on SAYAC embedded system.

SAYAC SUI enables high-level simulation, debug and analysis of platforms containing processors and peripheral models. All models are generated in SystemC environment. The processor model is an instruction set simulator (ISS) and the peripherals are SystemC bus functional models. The SUI will be extended so that you can create your own platforms, new models of processors, and other platform components using SystemC libraries and run them in the simulation environment using the SUI. For the time of being, the executable file will be sufficient to load and execute a simulation and it is not necessary to write your own SystemC testbench / harness. This makes the system easy to use for the users who are not familiar with SystemC modeling.

### 7-1 SUI commands

In this section the general commands for running a simulation in the simulation environment will be explained:

The executable harness file is provided to the users in addition to the source files that are needed for a system simulation.

```
CD $SourceDirectory/SUI/Examples/minimalSystem
>ls
ISSPowerV2 keyboard Memory power Source system systemTest
SAYACsystem
```

To invoke the system under test the following command can be used:

```
>./SAYACsystem
```

After running this command argument, the user enters the SUI simulation environment:

```
Welcome to SAYAC system user interface
SUI>> Please enter your command
```

When exploring the source file in the Source.cpp file, you can see that it contains a main function:

```
int sc_main(int argc, char *argv[])
{
    sc_report_handler::set_actions (SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_,
    ...

    systemTest * TOP = new systemTest ("systemTest_TB");
    sc_trace_file* VCDFFile;

    VCDFFile = sc_create_vcd_trace_file("system_Main");
    sc_trace(VCDFFile, TOP -> clk, "clk");

    SoftwareUserInterface(TOP);

    return 0;
}
```

It is a normal cpp main() which provides access to the command line arguments passed into the executable using argc, and argv. Invoking the command line simulation control will be done by a call to:

```
SoftwareUserInterface(systemunder test *name)
```

- **Loading a program onto processor using command line**

The SUI command line allows a program to be loaded onto a processor instance in the design in two ways:

- You can load an assembly file of the program. In this option, the SAYAC assembler will generate the corresponding binary file and saves it in *binfile.txt*.
- You can load the binary file of the program.

For loading either of this option, the command ldm will be used, and each option is specified with its argument:

```
SUI>> ldm>> hlp>>

    -ldm -asm: First the assembler would generate the binary file and save
it in a file named binfile.txt and then load it into the memory

    -ldm -bin: For loading a binary file into the memory
```

- **Specifying the program start address**

The following command can be used to set the initial program counter for the processor.

```
SUI>> stl>> hlp>>
```

The starting location for the memory in order to read the instructions from there can be set to any number here by following syntax: (-stl -(number)).

```
SUI>> Please enter your command -stl 5
SUI>> stl>> program starting from location 5
```

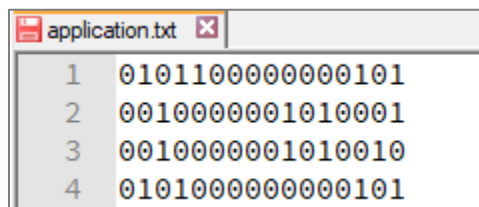
- **Generating memory initialization file using command line**

The user can generate the memory binary file using the command line arguments. Two options are available which both are similar in the syntax.

- The command *-gdf* generates the datafile. The syntax for generating a datafile is shown below. Note that for specifying the end of the file a dot should be inserted as the last argument of the *-gdf* command. The generated data file will be as the format shown in FIGURE 7.1.

```
SUI>> Please enter your command -gdf
SUI>> gdf>> enter the name of file you want to generate: application.txt
SUI>> gdf>> In order to generate a data file follow the following syntax:
address data

SUI>> gdf>> 0 01011000000000101
SUI>> gdf>> 1 0010000001010001
SUI>> gdf>> 2 0010000001010010
SUI>> gdf>> 3 0101000000000101
SUI>> gdf>> .
SUI>> Please enter your command
```



1	01011000000000101
2	0010000001010001
3	0010000001010010
4	0101000000000101

FIGURE 7.1 Generated data file

- The command *-wdf* generates the datafile and automatically loads the file into the memory. As shown in FIGURE 7.2 this command generates two separate data and address files with the names data.txt and address.txt respectively.

```

SUI>> Please enter your command -wdm
SUI>> wdm>> In order to generate a data file follow the following syntax:
address data

SUI>> wdm>> 0 01011000000000101
SUI>> wdm>> 1 0010000001010001
SUI>> wdm>> 2 0010000001010010
SUI>> wdm>> 3 0101000000000101
SUI>> wdm>> .
SUI>> Please enter your command

```

data.txt	
1	01011000000000101
2	0010000001010001
3	0010000001010010
4	0101000000000101

addr.txt	
1	0
2	1
3	2
4	3

FIGURE 7.2 Generated data and address files

- **Running the Simulation**

By using the following command, the system under test will be compiled and the program will be executed on the processor.

```

SUI>> Please enter your command -run
SUI>> run>> Running...

Init pow is done
Starting location: 0

Info: (1702) default timescale unit used for tracing: 1ps
(system_Main.vcd)

*****Guessed Password*****
Enter two character of the password          Time is: 20ns

```

- **Exiting the Simulation environment**

The following command ends the simulation and exits the simulation environment.

```

SUI>> Please enter your command -ext
user@user_name: ~/minimalSystem

```

- **Debugging using command line**

The following command enables the debugging mode for the simulation. Two arguments -on and -off are used for this command. When the user enables the debug mode, the user can trace and monitor the simulation. In this example, the power associated to each instruction in the software program can be traced and displayed.

```

SUI>> Please enter your command -dbg -hlp
SUI>> dbg>> hlp>> for debugging purpose:

        -dbg -on : entering the debugging mode
        -dbg -off: exiting the debugging mode

```

## 7-2 Example System

In this section one example system that is provided to SAYAC users is briefly described. The example is an embedded system that contains a processor, a memory, and a keyboard as a peripheral. FIGURE 7.3 shows the components in this system and the corresponding file hierarchy.

All the three modules are SystemC SC\_MODULES with member and main functions that model the functional behavior of each component. The models are bus functional models that are sensitive to the clock signal and consider the timings.

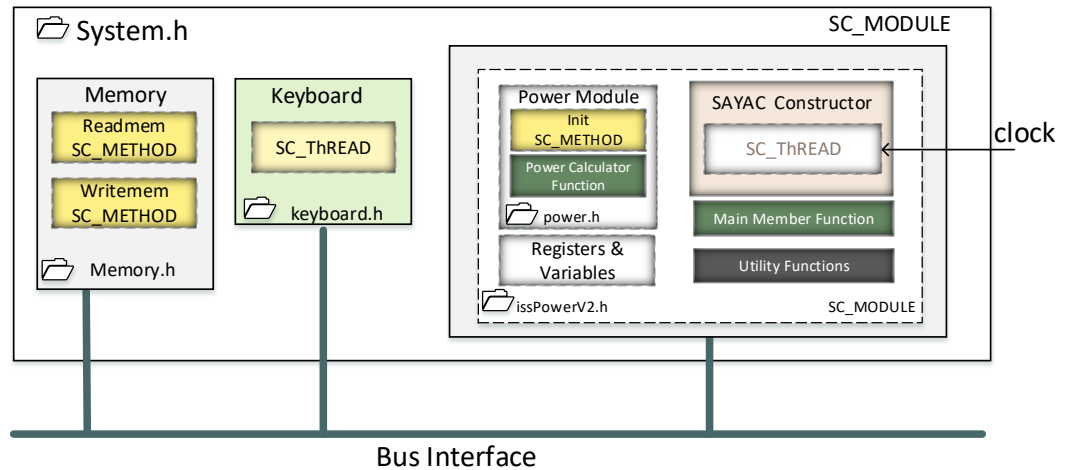


FIGURE 7.3 Embedded system example

- **SAYAC Processor ISS:**

The processor is modeled as a cycle accurate Instruction Set Simulator. FIGURE 7.4 shows the structure that is used in the process ISS model.

- **Ports and Constructors:**

The ports of SAYAC are of System C *sc\_signal* and include all the top-level inputs and outputs as shown in FIGURE 7.5.

The constructor includes a **SC\_THREAD** process that is sensitive to the clock.

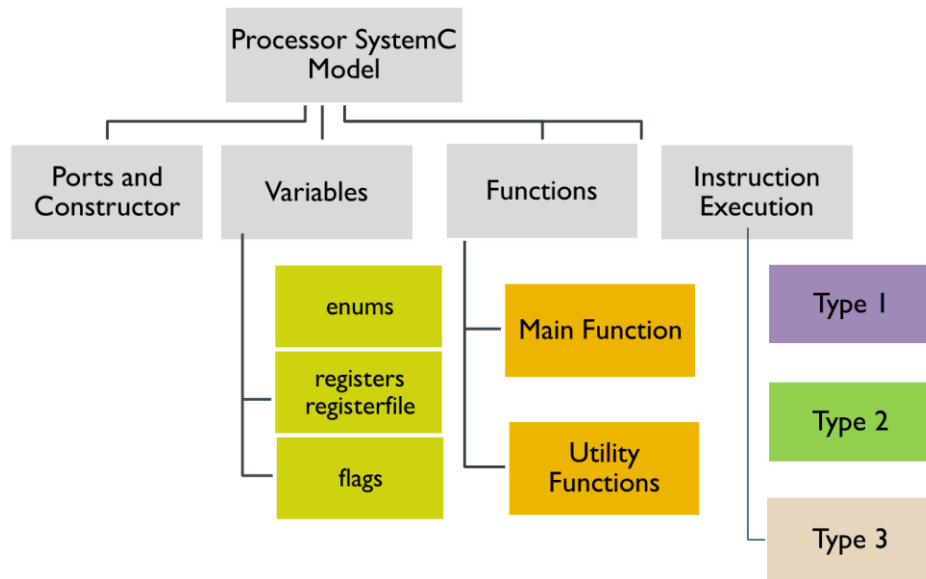


FIGURE 7.4 Embedded system example

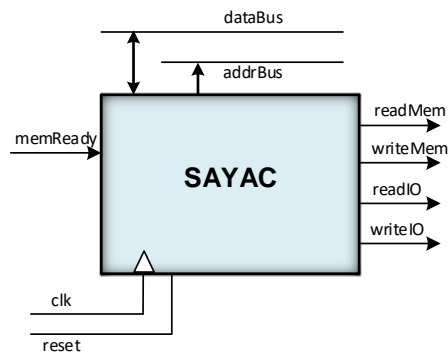


FIGURE 7.5 SystemC ports

### ➤ Variables:

The SAYAC ISS has the following variables:

- enums: enumeration types are used to define the opcodes of instructions. This helps to categorize instructions with the same attributes.
- The Register file (*regfile*): The register file is defined as an array of 16-bits *sc\_lv* variables. No hardware implementation is considered for modeling the registers and the register file. This improves the simulation speed significantly in comparison to a complete RTL implementation. Generic registers like the PC are 16-bit *sc\_lv* variables.
- The required flags for ALU or control instructions are all variables.



### ➤ Functions:

Two types of functions are used in this ISS. One is the main *abstractSimulation* function that performs the fetch, decoding, and execution. The main function is a switch-case statement that based on the opcode variable decides on the operation. Executing the instructions will be completed by the aid of some utility functions. Utility functions in SAYAC ISS are as follows:

- *WriteRegfile*: Function for writing into the registers of register file
- *ReadRegfile*: Function for reading from the registers of register file
- *nBitSignExtension*: Function for sign extension of the immediate
- *ShiftFunction*: Function for shifting to the right or left

### ➤ Instruction Execution:

To increase the speed of decode and execution, the instructions are categorized in three types. Each type of instructions have the same attributes in case of their opcodes.

Instruction	Notation
<b>Type 1</b>	
LDR	.rd. <= [.rs1.]
STR	[.rd.] <= .rs1.
JMR	PC <= PC + .rs1.
JMI	PC <= PC + 1 if s = 1
	PC <= PC + "imm"
	rd. <= PC + 1
<b>Type 2</b>	
ANR	.rd. <= .rs1. AND .rs2.
ANI	.rd. <= .rd. AND USE"imm"
MSI	.rd. <= SE"imm"
MHL	.rd. 'MSB <= "imm"
SIR	.rd. <= .rs1. LS±.rs2.
SAR	.rd. <= .rs1. AS±.rs2.
ADR	.rd. <= .rs1. + .rs2.
SUR	.rd. <= .rs1. - .rs2.
ADI	.rd. <= .rd. + SE"imm"
SUI	.rd. <= .rd. - SE"imm"
MUL	.rd. <= .rs1. × .rs2. 'LSB
	.rd+1. <= .rs1. × .rs2. 'MSB
DIV	.rd. <= .rs1. ÷ .rs2. 'Quo
	rd+1. <= .rs1. ÷ .rs2. 'Rem
<b>Type 3</b>	
CMR	flags <= Cmp(.rs1., .rd.)
CMI	flags <= Cmp(.rd., SE"imm")
BRC	PC <= .rd. if flag
BRR	PC <= PC + .rd. if flag
SHI	.rd. <= .rd. LS± "shim"
	.rd. <= .rd. AS± "shim"
NTR	.rd. <= 1sComp(.rs1.)
	.rd. <= 2sComp(.rs1.)
NTD	.rd. <= 1sComp(.rd.)
	.rd. <= 2sComp(.rd.)

**Nomenclature:**  
 .rsd. → R<sub>i</sub> content pointed by rsd  
 [ adr ] → Memory addressed by adr  
 ( loc ) → IO device addressed by loc  
 SE"imm, USE"imm → Signed, Unsigned Extension of "imm"  
 'LSB → Least Significant Byte  
 'MSB → Most Significant Byte

FIGURE 7.6 Instruction categorization

FIGURE 7.6 shows the type of instructions:

- *Type1*: Includes load, store and jump instructions. The opcode attribute of this type is 0010.
  - *Type2*: Includes the arithmetic and logical operations with the opcode attribute range of 0011-1110.
  - *Type3*: Mostly include control instructions and the opcode field is 1111.
- **SAYAC memory model:**

The memory model in the SAYAC system consists of two SC\_THREAD processes for reading and writing. These processes are sensitive to the signal *readMem* and *WriteMem* respectively. The utility functions in the memory model are as follows:

- *writememr*: Function for writing data to the memory
- *Readmemr*: Function for reading data from the memory
- *init*: This function controls the simulation based on the initialization commands or the default case. The data file will be loaded to the memory at the beginning of the execution.
- *dump*: Function for dumping memory values to an output file
- *setMemready*: For setting the signal *memready* to 1 when memory is ready

- **SAYAC keyboard module:**

The keyboard module is a SystemC module with one SC\_THREAD process that is sensitive to the *readIO* signal. This module receives two characters from the user. These characters will be used for password checking.

### 7-3 Example System Command Line Results

In this section the example system described above will be used to run a hardware security application on SAYAC processor. First the application program will be described and then the results of the program in SAYAC SUI will be presented.

- **Hardware Security Program:**

In some specific APPs, in order to access the provided services, a user has to first grant access, providing a correct predefined password.

- Passwords must be 8 characters long;
- APP performs the password correctness checking exploiting a custom co-processor, named SHADOW;
- SHADOW is implemented resorting to the SAYAC processor;
- APP and SHADOW interact via a shared memory called *Mem*, and specifically:

■ To request a new password check, APP has to:

- store the 16 lower bits of the password into *Mem*(0x0F00),
- store the 16 higher bits of the password into *Mem*(0x0F01),
- store the value x000F into *Mem*(0x0F02);
- at the completion of its check, SHADOW stores into *Mem*(0x0F10):
  - the value x0000 if the check failed,
  - the value x000F if the check passed;

In addition, the designers of SHADOW erroneously used a debug-oriented version of the processor SAYAC. In such a version, 2 general purpose registers (namely, R3 and R4) are used to store, at the completion of each Von-Neumann cycle, the number of clock cycles elapsed from the beginning of the current password checking, and the estimated power consumption from the beginning of the current password checking, respectively.

- **Hardware Security Program:**

For running the program on the SAYAC processor, the only arguments that user needs to use is the *-run* command in the SUI. For simulating the scenario that the user enters debug mode, one can use command *-dbg* first and then run the program.

```
SUI>> Please enter your command -dbg -hlp
SUI>> dbg>> on>> You entered the Debugging mode
*****
SUI>> Please enter your command -run
SUI>> run>> Running...
Init pow is done
Starting location: 0

Info: (I702) default timescale unit for tracing : 1 ps (system_Main.vcd)
*****MSI Instruction*****
[adr] 5
Regfile Write Data Is: 1111111110000000
#####Program Begins #####

Power is: 6
Time is: 5 ns

*****Ldr Instruction*****

*****Guessed Password *****
Enter two character of the password Time is 20ns

fr
[adr] 1
Regfile Write Data Is: 0110011001110010
Power is: 31
Time is: 25 ns
```

FIGURE 7.7 Running the program in debug mode

FIGURE 7.7 shows a part of simulation results in debugging mode. For each instruction there is the option of tracing the processor registers. Two features, power and time are also available in debugging mode. For each instruction the power will be accumulated and displayed. In addition, the simulation time will be displayed. At the end of program a message will be displayed as shown in FIGURE 7.8 indicating the total power of running program.

```
#####The Final Accumulated Power #####
```

```
Power is:4271
```

FIGURE 7.8 Total power dissipation

## **Index**