Table of contents

Table of contents 1

Transactions multiplexer. 4

Internal tag of Base Execution Unit. 5

Internal tag of ASR Portal...5

Internal tag of FPU. 5

Stream controller’s tag. 6

Tags of Context controller and Messenger. 6

Allocation of physical address space. 6

BootSRAM module. 7

Main memory space. 9

SDRAM main memory mode. 9

Internal SSRAM main memory mode. 9

ISI main memory mode. 9

Base Execution Unit. 12

General Purpose Registers...14

Data processing units. 14

Prefetcher. 14

Sequencer. 15

Data distribution circuit. 17

Address Translation Unit. 17

Stream controller. 17

Context Controller. 17

Messenger. 23

Frame Processing Unit – FPU. 24

Routing Engine – RTE. 25

Application-specific resource. 26

Instruction subsection diagram. 30

Context store diagram. 31

Context load diagram. 31

Messages subsection diagram. 32

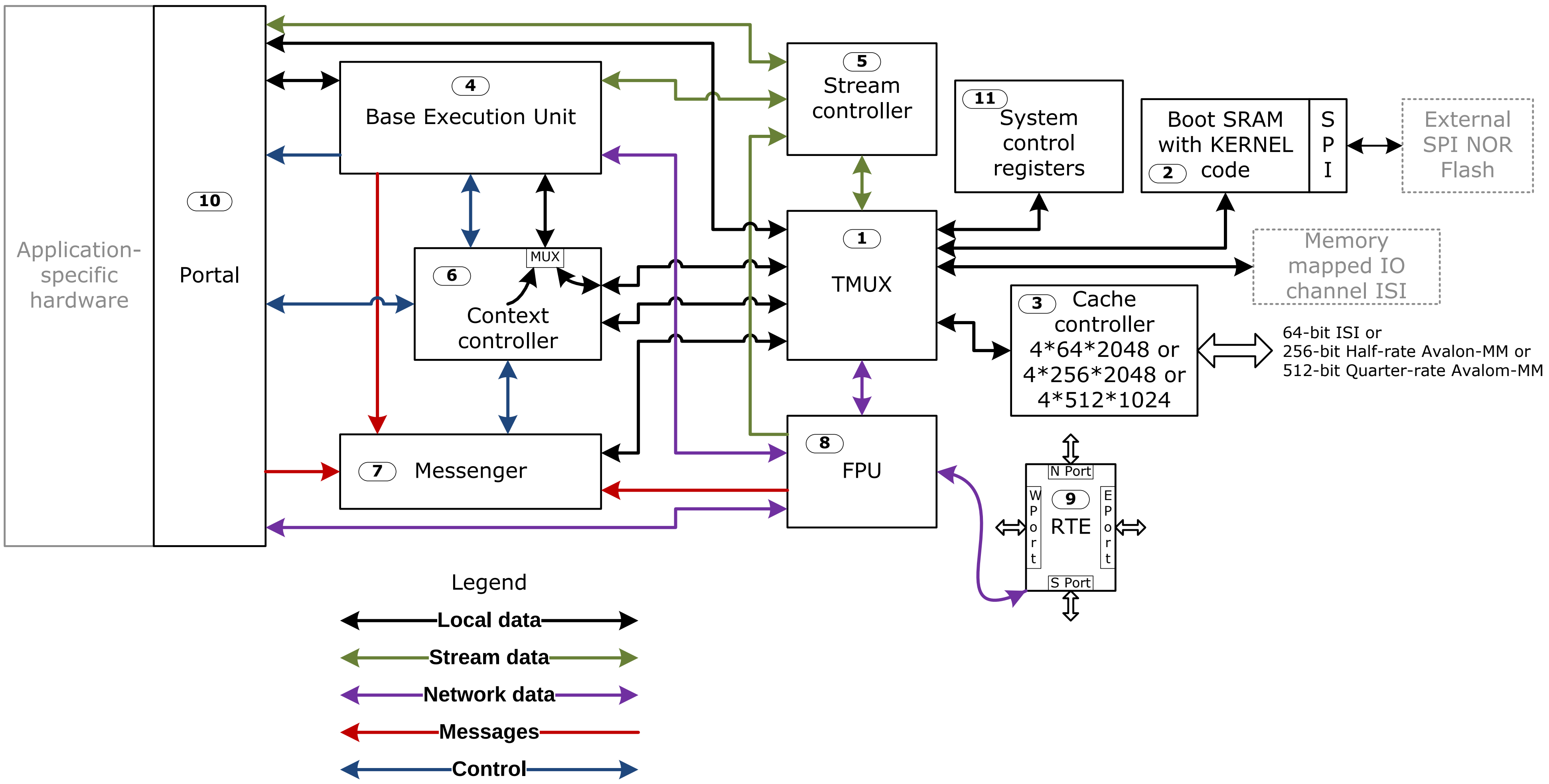
Memory interface subsection diagram. 33

System registers. 34

X32Carrier processor architecture.

The basic core configuration consists of the following blocks.

1. Transaction multiplexer — TMUX.
2. BootSRAM Module.
3. Main memory provided as an option:
   - SSRAM module;
   - Cache module and SDRAM controllers;
   - In-System Interface.
4. Basic Execution Unit containing:
   - Instruction flow unit - Prefetcher;
   - Instruction Manager - Sequencer;
   - General purpose registers file;
   - Floating-point adder, vector adder;
   - Floating-point multiplier, vector multiplier and integer multiplier;
   - Floating-point divider, vector divider and integer divider;
   - Integer ALU;
   - Parallel shifter;
   - Miscellaneous instructions unit;
   - Read/Write instructions channel and address translation unit.
5. Stream controller.
6. Context controller.
7. Message passing controller - Messenger.
8. Block for processing multiprocessor transactions – Frame Processing Unit (FPU).
9. Routing block of multiprocessor transactions – Routing Transactions Engine.
10. Portal used to connect Application-Specific Resource (ASR) to the base core. Portal contains channels:
    - Application-specific instructions;
    - Access channel to basic general purpose registers;
    - Loading/unloading context of application-specific registers;
    - Data transfer channel between application-specific resource and memory subsystem.
    - Channel for sending messages.
11. A set of system control registers.


Application-specific hardware
10
Portal
4
Base Execution Unit
5
Stream controller
11
System control registers
Boot SRAM with KERNEL
2
code
S P I
External SPI NOR Flash
1
TMUX
Memory mapped IO channel ISI
3
Cache controller
4*64*2048 or 4*256*2048 or 4*512*1024
64-bit ISI or
256-bit Half-rate Avalon-MM or
512-bit Quarter-rate Avalom-MM
6
MUX
Context controller
7
Messenger
8
FPU
N Port
W Port
9
RTE
E Port
S Port
Legend
Local data
Stream data
Network data
Messages
Control

Transactions multiplexer.

This block serves memory read and write requests from 6 devices of processor's core /written in descending order of service priority/:

- Base execution unit channel;
- Application-specific resource (ASR) Portal channel;
- Multiprocessor Frame Processing Unit (FPU) channel;
- Stream controller channel;
- Context controller channel;
- Messages processing unit (Messenger) channel.

The multiplexer processes all transactions in a pipelined mode and different memory devices can return read data with different time delays. This can change the order in which data arrives at read transaction initiators. To determine the ownership of a transaction, the transaction multiplexer generates a complete 21-bit transaction system tag. This tag must accompany the data being read from the memory device.
The figure below shows the format of the transaction tag.

20 18	17	15 13	12 0
Address [2:0]	Size	Channel number	Channel’s internal tag

Bits [20:18] copy the least significant 3 bits of the address. Bits [17:16] contain the operand's bit width code.

Size	***Description***
0	8-bit
1	16-bit
2	32-bit
3	64-bit

Together, the Address and Size fields allow you to define the data bus lines from which the operand must be received in read transactions and placed on the internal bus starting from bit 0. For example, if Address=2 and Size=1, then the transaction multiplexer extracts a 16-bit value from lines [31:16] of the input bus and places it on lines [15:0] of the internal data bus.

Channel Number - defines the channel number to which the transaction multiplexer should send the data read from the memory subsystem.

Channel Number	***Destination***
0	Base Execution Unit registers.

Channel Number	Destination
1	ASR Portal.
2	Multiprocessor Frame Processing Unit channel.
3	Stream controller channel.
4	Context controller channel.
5	Messenger channel.
6, 7	Reserved.

Internal tag of Base Execution Unit.

Local tag [12:0]	Register
001Fh – 0000h	Bits [63:0] of General-Purpose Registers R[31:0]
003Fh – 0020h	Bits [127:64] of General Purpose Registers R[31:0]
005Fh – 0040h	Flag registers AFR[31:0]
006Fh – 0060h	Address registers AR[15:0]
011Fh – 0100h	Descriptor cache registers DTR[7:0]
0140h	Loading program code into the instruction cache.
0144h	Loading program code into the instruction cache, the last word.
0149h	Loading a new value into the instruction pointer.
0150h	Loading program code in violation of code object limit.
0154h	Loading program code into the instruction cache, the last word. Code object limit violation detected.

Internal tag of ASR Portal.

Local tag [12:0]	Register
00FFh – 0000h	ASR registers defined as a switchable resource context.
011Fh – 0100h	Descriptor Cache Registers. Portal has 8 descriptor cache registers.

Internal tag of FPU.

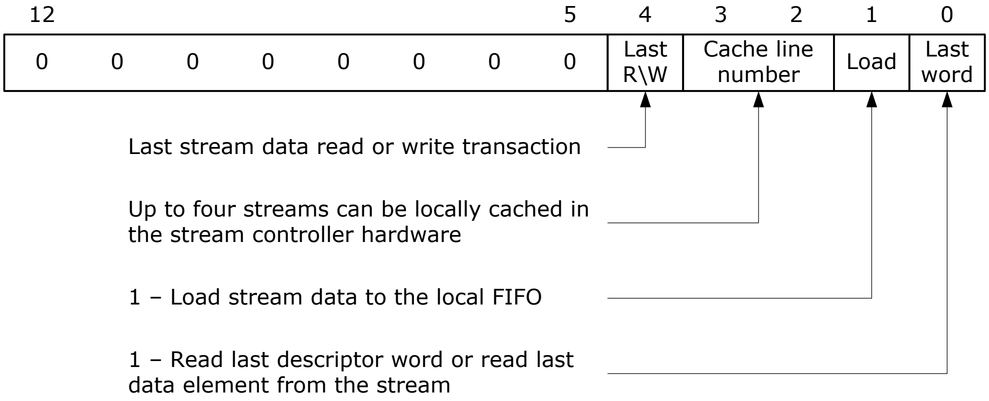
12	11 4	3 0
0	Core number	Network transaction tag

This tag is used by the multiprocessor transaction block to identify the core to which the data read from local memory should be returned and to indicate which transaction within this core the data corresponds to.

12	11	10	4	3	2	1	0
1	0	Descriptor cache index		0	0	Word Num	

The tag is used to load descriptors into the local descriptor cache of a FPU. The cache is designed for 128 object descriptors, the cache index is specified in bits [10:4] of the tag. The lower 2 bits determine the number of the 64-bit descriptor word to be loaded.

Stream controller’s tag.



12							5	4	3	2	1	0
0	0	0	0	0	0	0	0	Last R\W	Cache line number		Load	Last word

Last stream data read or write transaction

Up to four streams can be locally cached in the stream controller hardware

1 – Load stream data to the local FIFO

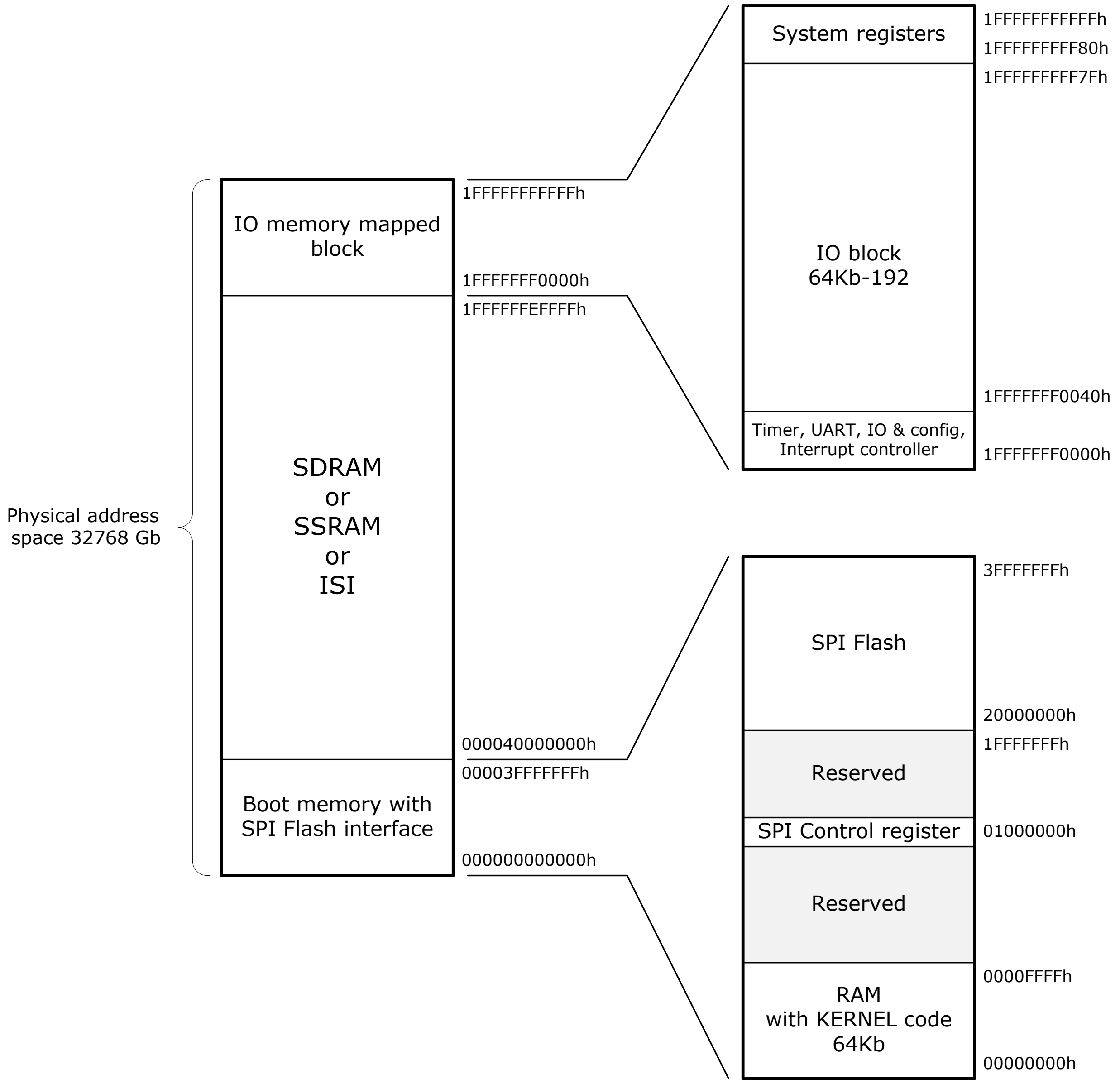
1 – Read last descriptor word or read last data element from the stream

Tags of Context controller and Messenger.

These controllers do not use a transaction tag, their transaction tags are always 0. The specifics of these controllers is that they always wait for the requested data to be received from memory before initiating the next read transaction. Therefore, there is no need to identify the internal ownership of the read data and there is no need to use a tag.

Allocation of physical address space.

The figure below shows the allocation of the processor's physical address space.



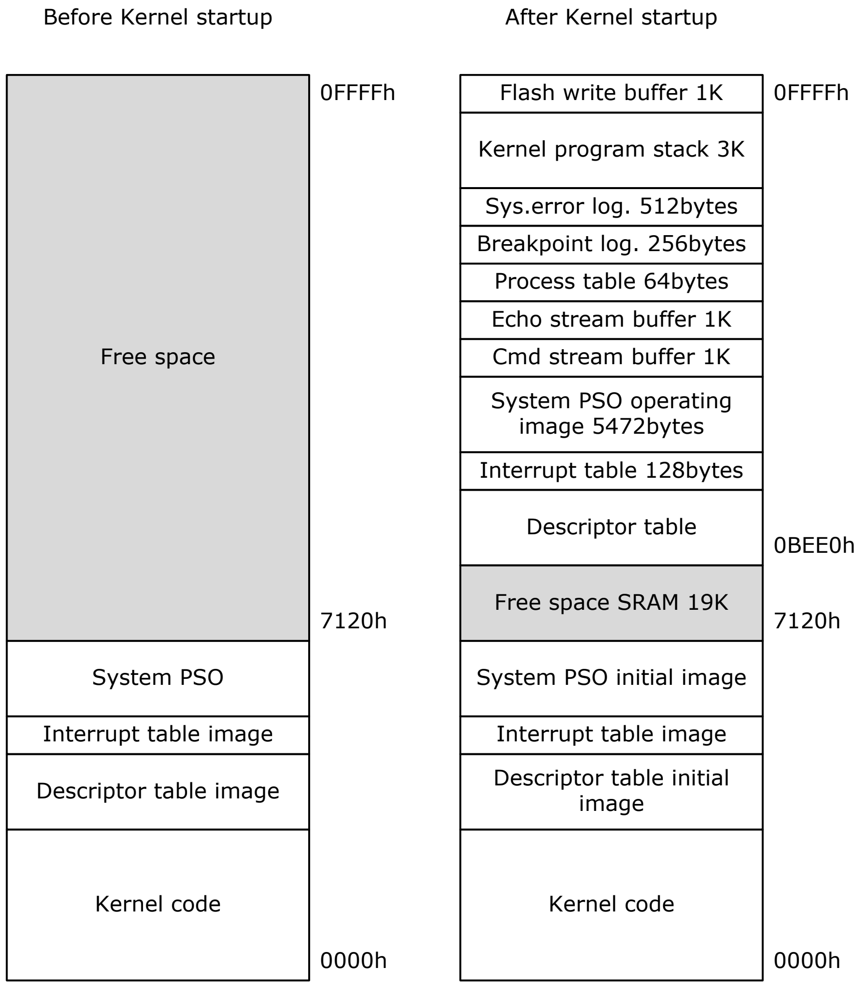
The lower 64K is the built-in RAM, which also contains the KERNEL system software code.

Reading data from an external SPI Flash is carried out at addresses in the range from 20000000h to 3FFFFFFFh. Reading an array can be done with regular machine instructions for loading data from memory, but the SPI control register must contain an array read command code, the default value of which is 0Bh.

Starting from the address 000040000000h and 1FFFFFFEFFFFh, the transaction multiplexer accesses the main RAM space. It can be like an SDRAM controller with a cache system, it can be a static memory buffer implemented on internal FPGA memory blocks, the In-System Interface (ISI) option is also possible, to which you can connect any other specific memory subsystem.

BootSRAM module.

The module contains a memory buffer with a capacity of 64K and an organization of 8K*64. This buffer contains the KERNEL system software code and initial images of the descriptor table, interrupt table, and system process state object (PSO). When starting the Kernel after a system reset or restarting when a fatal system error occurs, the system structures are expanded, as shown in the figure below.



The module also contains an SPI controller to which you can connect an external SPI Flash memory chip. The controller has a 32-bit control register located at physical address 01000000h. Its format is shown in the figure below.

31 24	23 16	15 10	9 0
X X X X X X X X	Instruction for read operations	X X X X X X	Bytes count

The lower 10 bits of the register are used to indicate the count of bytes that are in the Flash write buffer and that must be transferred to Flash. The instruction code byte used when reading data from Flash is located in bits [23:16] of the control register. Writing the two most significant bits of the counter [9:8] starts the Flash controller to transfer the specified number of bytes located in the Flash write buffer /physical address 00FC00h/ to the Flash chip. For example, to write 100 bytes to Flash, you need to do the following:

- Place the 02h write command byte in the Flash write buffer.

- After the write command byte, 3 bytes of the write address of the first data byte must be marked.
- Following the address, 100 bytes of data must be written.
- In bits [9:0] of the control register put the number 104 /100 bytes of data + 3 bytes of address + command byte/ Only after that the controller will transfer to Flash a sequence of 104 bytes.

To unlock write protection in Flash you need:

- Put the code 06h into the write buffer.
- Write the data length to the control register - 001h.

To read data from Flash you need:

- In bits [23:16] of the control register, put the data read command code - 0Bh.
- And then read data from the Flash address space with the memory read instructions with any addresses.

To read the status register of the Flash chip:

- In bits [23:16] put the command code for reading the status register - 05h.
- Read a byte at any address belonging to the Flash space.

Main memory space.

SDRAM main memory mode.

The connection of the core to the SDRAM controller is always done through the cache controller. There are 3 types of cache controllers available:

1. Cache4x64x2048 - 4 channels with a capacity of 2K * 64 each with an external 64-bit ISI bus designed to connect a pre-defined type of physical interface to the memory.
2. Cache4x256x2048DDR3 - 4 channels with a capacity of 2K * 256 each and an external Avalon-MM interface with a bit width of 256 bits, a packet length of 4, used to connect to the DDR3 SDRAM Controller with UniPHY IP module with Half-rate mode.
3. Cache4x512x1024DDR3 - 4 channels with a capacity of 1K * 512 each and an external Avalon-MM interface with a bit width of 512 bits, a packet length of 4, used to connect to the DDR3 SDRAM Controller with UniPHY IP module with Quarter-rate mode.

Internal SSRAM main memory mode.

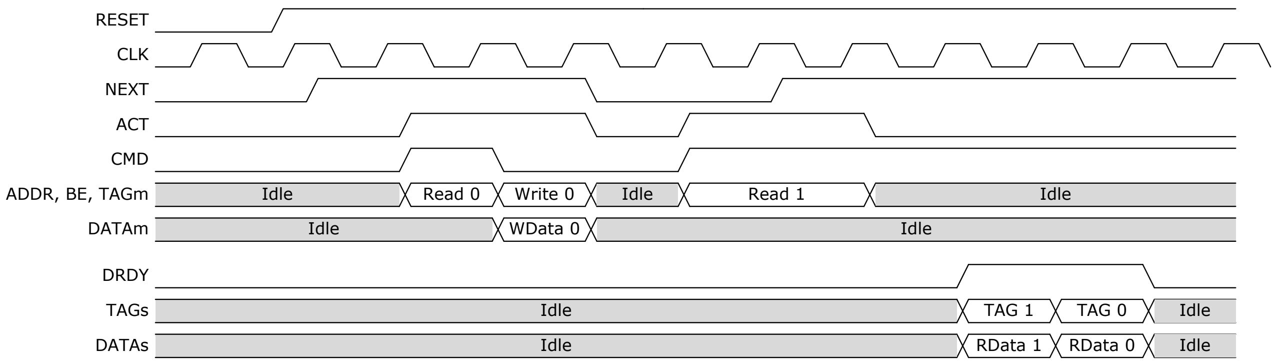
4 internal SSRAM options are available: 256Kb, 512Kb, 1Mb, 2Mb. FPGA block memory blocks are used to build SSRAM.

ISI main memory mode.

The ISI variant can be used when it is assumed that the RAM space will contain different types of memory modules. In this case, additional transaction demultiplexing and read data multiplexing are required. An additional transaction multiplexer in this case is connected via ISI to the core transaction multiplexer. The table below shows the composition and description of the lines of the ISI interface.

Bus	***Direction***	***Description***
CLK	-	Main clock for ISI-Master and ISI-Slave.
RESETn	-	Main RESET for ISI-Master and ISI-Slave. Active – 0.
NEXT	M←S	Slave is ready to accept a new transaction in the current cycle when NEXT=1. NEXT=0 – the master must hold the transaction parameters until the end of the next cycle.
ACT	M→S	Transaction activation.
CMD	M→S	CMD=1 – read transaction activation, 0 – write activation.
ADDR[44:0]	M→S	Address, specifies the address of the low byte of the data element to be transmitted.
BE[7:0]	M→S	The bus defines the active bytes on the data lines in write transactions. The active level is 0. For example, BE=0F3h defines the transfer of two bytes over the DATA[31:16] lines.
DATAm[63:0]	M→S	Data to be written to a memory location or register of a Slave device.
TAGm[20:0]	M→S	Transaction tag. Used in read transactions to determine the destination of data within the processor’s core. The slave device must return the transaction tag along with the retrieved data to the master device.
DRDY	M←S	data ready. The readiness of the data requested in the read transaction.
DATAs[63:0]	M←S	Read data.
TAGs[20:0]	M←S	Read transaction tag.

Diagrams of ISI work.



The data returned by the slave device may not be returned in the order in which the corresponding requests were received from the master. But that doesn't matter, because in the core, data ownership is determined by the transaction tag.

Placement of the written data on the DATAm bus depending on the address and data width.

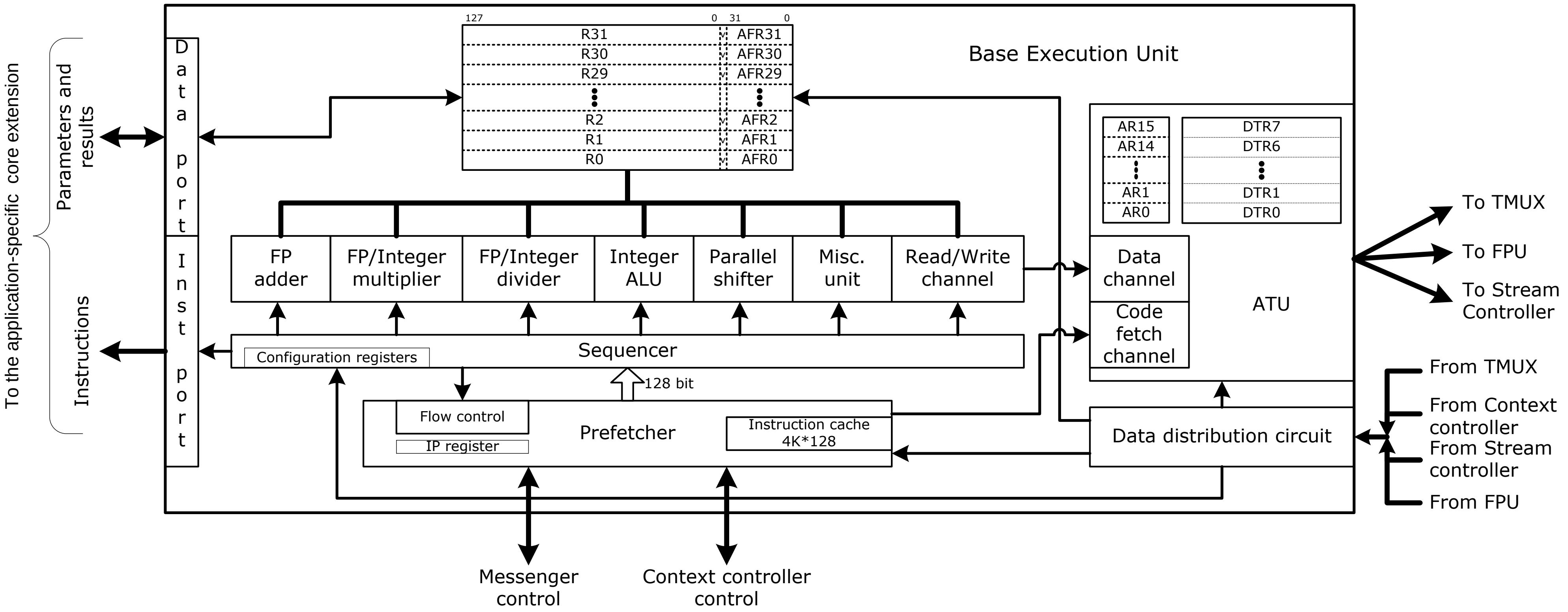
ADDR[2:0]	***BE[7:0]***	***DATAm lines state***
Byte width		
0	0FEh	63 56 55 48 47 40 39 32 31 24 23 16 15 8 7 0 Data byte Data byte Data byte Data byte Data byte Data byte Data byte Data byte
1	0FDh	
2	0FBh	
3	0F7h	
4	0EFh	
5	0DFh	
6	0BFh	
7	7Fh	
Word width		
0,1	0FCh	63 48 47 32 31 16 15 0 Data word Data word Data word Data word
2,3	0F3h	
4,5	0CFh	
6,7	3Fh	
DWord width		
0-3	0F0h	63 32 31 0 Data Dword Data Dword
4-7	0Fh	
QWord width		
0-7	00h	63 0 Data Qword

Placement of data on the DATAs bus as it is received from the slave.

ADDR[2:0]	***Data location on the DATAs lines***
Byte width	
0	DATAs[7:0]
1	DATAs[15:8]
2	DATAs[23:16]
3	DATAs[31:24]
4	DATAs[39:32]
5	DATAs[47:40]
6	DATAs[55:48]
7	DATAs[63:56]
Word width	
0,1	DATAs[15:0]
2,3	DATAs[31:16]
4,5	DATAs[47:32]
6,7	DATAs[63:48]
DWord width	

ADDR[2:0]	***Data location on the DATAs lines***
0-3	DATAs[31:0]
4-7	DATAs[63:32]
QWord width	
0-7	DATAs[63:0]

Base Execution Unit.


To the application-specific core extension
Parameters and results
Instructions
Data port
Inst port
127
0
31
0
R31
R30
R29
R2
R1
R0
AFR31
AFR30
AFR29
AFR2
AFR1
AFR0
Base Execution Unit
AR15
AR14
AR1
AR0
DTR7
DTR6
DTR1
DTR0
FP adder
FP/Integer multiplier
FP/Integer divider
Integer ALU
Parallel shifter
Misc. unit
Read/Write channel
Data channel
Code fetch channel
ATU
To TMUX
To FPU
To Stream Controller
Sequencer
Configuration registers
128 bit
Flow control
IP register
Prefetcher
Instruction cache 4K*128
Data distribution circuit
From TMUX
From Context controller
From Stream controller
From FPU
Messenger control
Context controller control

General Purpose Registers.

BEU Contains 32 general purpose registers. Associated with each general purpose register is a register of operation result flags. These flags are written to the AFR at the same time as the result is written to the appropriate general purpose register. Each general purpose register has a valid flag indicating that the contents of the register are ready to be used as a source operand, or the register is ready to accept new data.

Data processing units.

There are 6 main data processing units.

1. A floating point adder that performs addition and subtraction operations.
2. Multiplier.
3. Divider.
4. Integer ALU.
5. Parallel shifter.
6. Miscellaneous operations unit.
7. The channel for data read/write operations and data exchange with address registers.

Floating point processing blocks (adder, multiplier, divider) work with four data formats:

- Single-Precision (32 bit);
- Double-Precision (64 bit);
- Extended-Precision (128 bit);
- Vector data.

In turn, vector data can be of two types:

- A single-precision vector, when four single-precision numbers are packed into a 128-bit word;
- A double-precision vector when two double-precision numbers are packed into a 128-bit word.

The divider also implements the calculation of the square root. The operation of extracting the square root is possible only on floating point numbers. The divider and multiplier are also capable of handling signed or unsigned integer data.

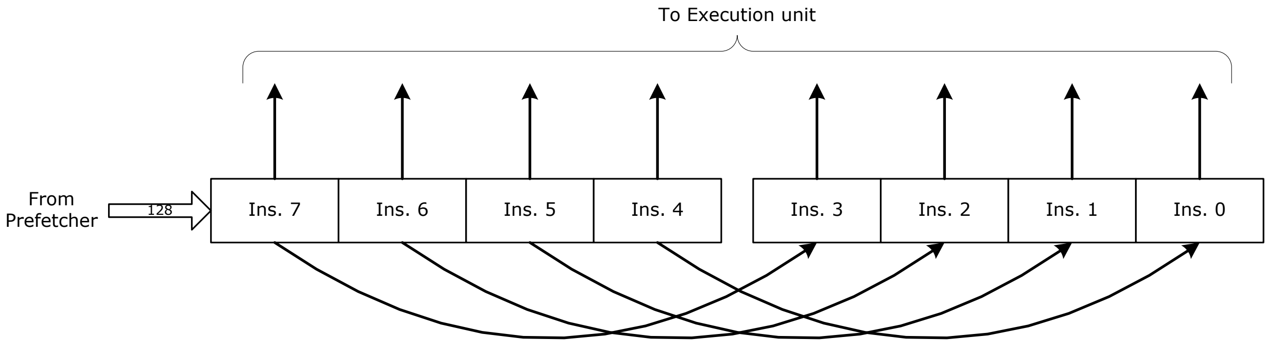
Prefetcher.

Prefetcher pumps the instruction cache from external memory and forms a 128-bit stream that transfers 4 instructions per cycle to the Sequencer. The instruction cache is cleared each time the current code object changes, either as a result of a process-to-process switch or a transfer of control to a message handler. Prefetcher generates read requests to the address translation unit when loading the instruction string into the cache. The line length is 32 64-bit words, which corresponds to 64 instructions. From the Sequencer, the Prefetcher receives a stream of control transfer instructions within the current code object, as well as instructions for allocating memory, reading a message parameter, and transmitting a message. These instructions cause the corresponding requests to be generated in the Context Controller and in the Messenger.

Sequencer.

Sequencer analyzes the readiness of instructions for execution depending on the readiness of their source operands and the result receiver. The readiness of the registers involved in the operation is determined by a special 32-bit flag register. Each bit of this register indicates that the corresponding general purpose register is ready for use. Sequencer performs competitive execution of instructions. The order in which instructions are executed may not match their sequence in the program. This is true for almost all types of instructions except control transfer instructions, instructions for transferring data between memory and internal registers, and transferring data between general purpose registers and address registers. The data transfer and control transfer sequences are carried out strictly as specified in the program.

The sequencer has a 256-bit instruction register, into which 4 instructions are loaded from the Prefetcher. Instructions are stored in this register until they are ready to be executed when the source operands are ready or the result destination is ready.



Situations are possible when all 8 instructions from the instruction register will be transferred for execution to the execution unit simultaneously on one cycle, if they do not depend on each other in terms of operands and result receivers. Up to 4 instructions can be executed at the each clock cycle at the same time, as they come from Prefetcher. Prefetcher fills the register with instructions [7:4]. When the instruction register [3:0] is empty, instructions [7:4] that have not yet been executed are transferred to positions [3:0], and positions [7:4] are loaded with a new set from the Prefetcher.

The dependencies of instructions on the readiness of the source operands and the result receiver are determined using special configuration registers located in the Sequencer and initialized at the start of the processor from a special configuration ROM located in the Context Controller. Configuration registers are 256-bit. Each bit determines what the Sequencer does with a specific instruction field, depending on the 8-bit instruction code. There are 7 configuration registers in total, containing the following 256-bit vectors (in parentheses are the base values that support the basic instruction set).

255 0

PortalVector /0000000000000000000000000000000000000000000000000000000000000000/
SkipVector /0000000000000000000000000000000000000000000000000000000000000000/
Op2Vector /00000000000000FF000000000000023300000000002F7FFF000000000DEFFFFF/
Op1Vector /0000000000000000000000000000110300000000000000800000000005EFFFFF/
Op0Vector /000000000000000000000000000000000000000000002F7BFF000000000DEFFFFF/
NopVector /FFFFFFFFFFFFC000FFFFFFFFFFFFE00CFFFFFFFFFF008000FFFFFFFFF0000000/
InvdVector /FFFFFFFFFFFFFF3DFFFFFFFFFFFFFDDDFFFFFFFFFF008000FFFFFFFFF0000000/

InvdVector determines which machine instructions will modify the general purpose register. An instruction that modifies a general-purpose register resets the register's validity flag when it is started for execution. The zero bit of the vector indicates that the corresponding instruction sets the unavailability of the register-receiver of the result at the time of launch for execution. For example, all 28 instructions with codes from 00h to 1Bh place the results in the general purpose register and cause the receiver's register ready flags to be reset.

NopVector defines invalid opcodes that the Sequencer ignores. If a bit in NopVector is set to 1, then the corresponding opcode for that bit is an invalid instruction and the instruction is ignored.

Op0Vector tells the Sequencer to check if the register specified in bits [12:8] of the instruction can be used. If the Op0Vector bit is set to 1, then the Sequencer must check the readiness of the register indicated in bits [12:8], and also whether this register is used as a result receiver in instructions that have not yet been transferred for execution and are found in the text of the program previously checked instructions.

Op1Vector tells the Sequencer to check if the register specified in bits [20:16] of the instruction can be used as the operand source. The readiness of the register is checked, as well as its use as a result receiver in instructions located earlier in the program text, before the instruction being checked.

Op2Vector tells the Sequencer to check the register specified in bits [28:24] of the instruction to see if it can be used as a result destination. It can be used in instructions previously loaded into the Sequencer but not yet submitted for execution.

SkipVector is used only in instructions that work with the portal. The bits in this vector enable or disable checking that the application-specific resource is ready to receive a new instruction. If the vector bit is 0, then the instruction is considered transferred to the portal as soon as it is ready for execution. If the bit for an instruction is set to 1, then the Base Execution Unit, after sending this instruction to the portal, will not send any other instructions to it until it receives a signal to complete the current one. Thus, the vector allows you to separate the instructions of the application-specific resource into FlyBy and those whose execution must be waited for before submitting the next instruction to the portal.

PortalVector itself determines which instructions are instructions for an application-specific resource. A bit set to 1 indicates that the corresponding opcode belongs to an application-specific resource instruction.

Data distribution circuit.

The circuit distributes data read from the memory subsystem to recipients depending on the transaction tag. Multiplexes data from TMUX, the context controller, the flow controller, and the frame handler of the multiprocessor interface.

Address Translation Unit.

The unit contains 16 address registers AR[15:0] and 8 descriptor cache registers DTR[7:0]. Descriptor selector DTR0 is located in register AR1, descriptor DTR1 - in register AR3, DTR2 - AR5 and DTR7 - AR15 respectively. A descriptor register becomes invalid whenever a new object selector is written to its corresponding address register, and if the same value is entered into the register as it was before, the descriptor's valid flag is not cleared. This is done in order to suppress unnecessary descriptor loading procedures.

The address translation unit implements the following checks:

- Determining the transaction mode - local to an object in memory, local to a thread, or access to another core of a multiprocessor network.
- Checking offset for violation of object limits when accessing local objects.
- Checking whether an object can be accessed by privilege level, by whether the object is available for reading or writing, by whether the object belongs to the current process or group of processes.
- Support for object segmentation system. Finding an object segment and loading a descriptor of the object’s segment for which the offset specified in the logical address is valid.

Stream controller.

There are 2 types of objects: objects that are arrays of data bytes and stream objects. Both types are described by the corresponding access descriptors and are accessed via object selectors. Threads are virtual queues, similar to FIFO queues. Working with such objects is supported by special hardware called a stream controller. Each stream has bit width and length its main parameters. Each stream has its own representation in RAM in the form of a buffer, the size of which depends on the declared queue length and data width. The stream controller converts a regular array of RAM cells into a queue, saving the application using the streams from having to explicitly work with read/write pointers, check their mutual status, and simplifies the handling of situations of lack of data or overflow of the stream queue.

The stream controller is connected by data channels to:

- An application-specific resource portal that has both read and write access to streams;
- Basic execution unit, read and write;
- A frame processing unit that has access only to write data to the stream.

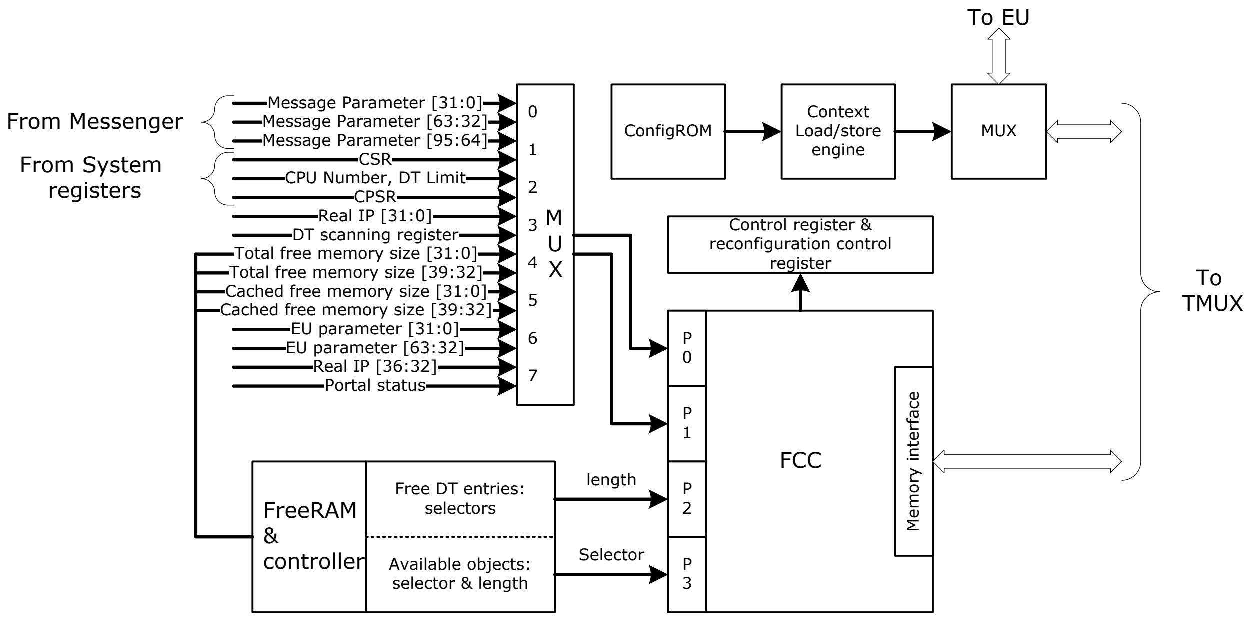
Context Controller.

The main purpose of the Context Controller is to manage the context of the processor's execution unit and, possibly, the context of an application-specific resource, if it provides the ability to change the context to serve different processes in time-sharing mode. The context is understood as information that is unique for each process running in the time-sharing mode on a common processor hardware resource for all processes. The context of processor execution units can be changed when processing a request from a process timer, when processing a hardware interrupt, or when processing a message.

The second function of the Context Controller is to configure the Sequencer before starting work, taking into account the features of the machine instruction set, extended by application-specific instructions.

The third function of the Context Controller is the processing of macro instructions for requesting memory blocks for creating objects and requests for deleting unnecessary objects.

The block diagram of the Context Controller is shown in the figure below.


To EU
From Messenger
From System registers
Message Parameter [31:0]
Message Parameter [63:32]
Message Parameter [95:64]
CSR
CPU Number, DT Limit
CPSR
Real IP [31:0]
DT scanning register
Total free memory size [31:0]
Total free memory size [39:32]
Cached free memory size [31:0]
Cached free memory size [39:32]
EU parameter [31:0]
EU parameter [63:32]
Real IP [36:32]
Portal status
0
1
2
3
4
5
6
7
MUX
ConfigROM
Context Load/store engine
MUX
Control register & reconfiguration control register
To TMUX
P 0
P 1
P 2
P 3
FCC
Memory interface
FreeRAM & controller
Free DT entries: selectors
Available objects: selector & length
length
Selector

The Context Controller contains a firmware-controlled controller /FCC/ in which algorithms for managing the execution unit, switching its context, algorithms for allocating and releasing memory blocks are implemented at the firmware level.

The FreeRAM buffer contains 256 56-bit cells. Designed to support the operation of the system for creating and deleting objects. Logically, it is divided into 2 halves. The lower half contains selectors corresponding to empty positions in the descriptor table. The table is designed to quickly find positions in the descriptor table, in which descriptors of created objects can be placed. The top 128 FreeRAM cells contain the free object selectors and their size in 32-byte paragraphs from which memory blocks can be sliced for new objects. The FreeRAM cache is updated periodically during memory allocation operations. With each program intervention in the contents of the descriptor table, KERNEL re-initializes the memory allocation subsystem by updating the FreeRAM context.

ConfigROM contains a configuration block that describes the portal settings and instruction set configuration.

Offset	*Description*
00h	InvdVector[31:0]
01h	InvdVector[63:32]
02h	InvdVector[95:64]
03h	InvdVector[127:96]
04h	InvdVector[159:128]
05h	InvdVector[191:160]
06h	InvdVector[223:192]
07h	InvdVector[255:224]
08h	NopVector[31:0]
09h	NopVector[63:32]
0Ah	NopVector[95:64]
0Bh	NopVector[127:96]
0Ch	NopVector[159:128]
0Dh	NopVector[191:160]
0Eh	NopVector[223:192]
0Fh	NopVector[255:224]
10h	Op0Vector[31:0]
11h	Op0Vector[63:32]
12h	Op0Vector[95:64]
13h	Op0Vector[127:96]
14h	Op0Vector[159:128]
15h	Op0Vector[191:160]
16h	Op0Vector[223:192]
17h	Op0Vector[255:224]
18h	Op1Vector[31:0]
19h	Op1Vector[63:32]
1Ah	Op1Vector[95:64]
1Bh	Op1Vector[127:96]
1Ch	Op1Vector[159:128]
1Dh	Op1Vector[191:160]
1Eh	Op1Vector[223:192]

Offset	***Description***
1Fh	Op1Vector[255:224]
20h	Op2Vector[31:0]
21h	Op2Vector[63:32]
22h	Op2Vector[95:64]
23h	Op2Vector[127:96]
24h	Op2Vector[159:128]
25h	Op2Vector[191:160]
26h	Op2Vector[223:192]
27h	Op2Vector[255:224]
28h	SkipVector[31:0]
29h	SkipVector[63:32]
2Ah	SkipVector[95:64]
2Bh	SkipVector[127:96]
2Ch	SkipVector[159:128]
2Dh	SkipVector[191:160]
2Eh	SkipVector[223:192]
2Fh	SkipVector[255:224]
30h	PortalVector[31:0]
31h	PortalVector[63:32]
32h	PortalVector[95:64]
33h	PortalVector[127:96]
34h	PortalVector[159:128]
35h	PortalVector[191:160]
36h	PortalVector[223:192]
37h	PortalVector[255:224]
38h	Control DWORD 38h

31	21		19 16	15 8	7			0
	SoE	ER	WSC	Context Length	CP	CF	REC	Base ISET

Offset	***Description***
39h	Control DWORD 39h 31–24 ▨ 23–16 Messages Counter 15–0 Messages Timer
3Ah	Control DWORD 3Ah 31–24 ▨ 23–16 GPR WDT 15–0 Portal WDT
3Bh	Control DWORD 3Bh 31–20 ▨ 19 PL ▨ 16 15–0 Task ID
3Ch	Control DWORD 3Ch 31–24 ▨ 23–0 Portal PSO
3Dh	Reserved
3Eh	
3Fh	Instruction set extension ID

InvdVector, NopVector, Op0Vector, Op1Vector, Op2Vector, SkipVector, and **PortalVector** are loaded into their respective Sequencer configuration registers at processor startup.

Base ISET Basic instruction set index, 4 bits. This field will be used in the future, when the processor configuration will change dynamically, and there will be several basic instruction sets. Currently there is only one version of the base set and its index is 1h.

REC the field defines who can reconfigure the resource:

- 0 - no one can, the resource is static, the value is currently in use.
- 1 - only a more privileged process than the RPL (Resource Privilege Level).
- 2 - only a process with the same TaskID as the resource or more privileged.
- 3 – any process.

CF. CF=0 - the resource is always stopped along with the base core when switching to another process and can be reconfigured if necessary. CF=1 - the resource can continue to work after a process change in the base core.

CP indicates the presence of the persisted context. If CP=1, then there is a saved context and its length is **Context Length**+1.

Context Length. Together with the CP bit, this field specifies the length of the application-specific resource context.

WSC A count of the wait cycles inserted into transactions generated by the resource. It is intended to limit traffic so that the resource does not block the system with permanent transactions. The number of idle cycles between transactions is WSC+1. Thus, there is always free time for system transactions.

ER Whether or not to allow in-memory access error messages. If ER=1, entry into the error queue of object access rule violations from the application-specific resource is allowed. If ER=0, then erroneous transactions are only blocked, but are not registered in the system error queue and do not cause the system error handler to run.

SoE Stop the resource when an error occurs, or continue working. If SoE=1, the resource is stopped when a system error is encountered.

Messages Timer A timer that controls how often messages are sent from an application-specific resource. This timer counts in 0.2 μs intervals. The interval countdown starts whenever the resource has sent a certain number of messages in a row.

Messages Counter The counter of the number of messages in the packet is 8 bits. For a certain period of time, a resource can generate a limited number of messages. The message counter is reset when the timer expires. Then again it will be possible to generate messages.

Portal WDT Portal busy watchdog timer – 2 bytes (wait for portal empty flag). The context controller stops the application-specific resource if it needs to change its context. To do this, a resource stop signal is generated. After that, the portal starts counting the timeout after which a resource stop signal is generated even if the resource itself has not issued the corresponding signal to the portal, which allows the context controller to continue the context change operation and not be blocked in an endless wait for the resource stop signal.

GPR WDT GPR busy watchdog timer – 1 byte. The timer counts down the number of 200ns intervals after which the GPR is restored as valid. This timer avoids situations of complete blocking of the processor if the application-specific resource does not write to the GPR. The timer will restore the GPR's validity and readiness for other operations.

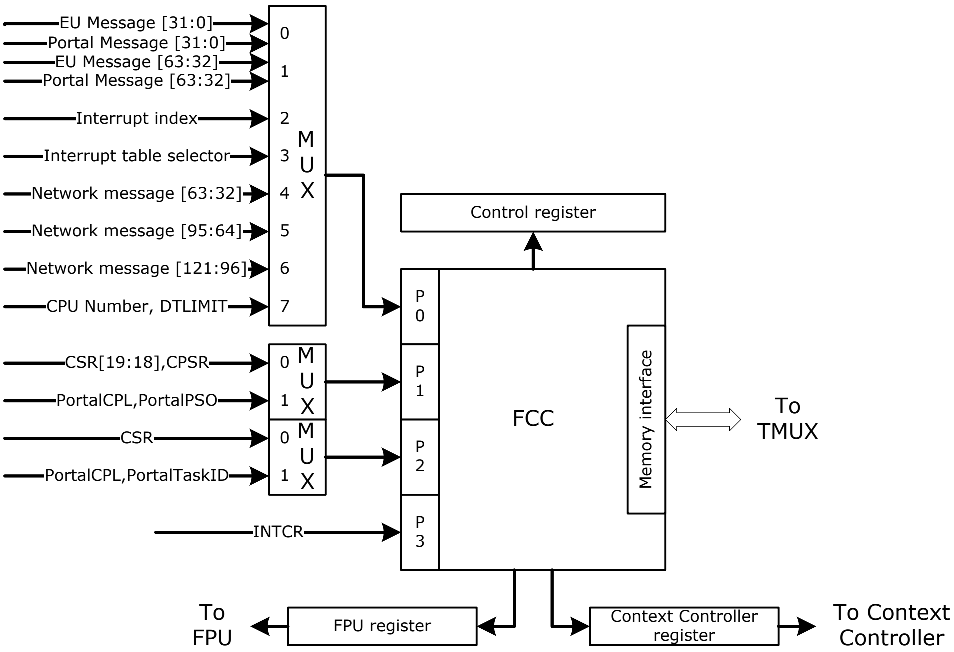
Task ID, PL, Portal PSO. These fields are not present in the context controller configuration ROM. But they are in the form of registers in the portal. The fields contain the current application-specific resource settings. The Task ID and PL are involved in checking whether the resource can access the object. Portal PSO - Specifies the PSO selector that hosts the resource context if the resource's architecture allows a context switch.

Messenger.

Messenger is used to resolve transfer issues on a global level. Its task is to determine which code of which process the processor will execute next. Decisions are made based on the presence of message transfer requests from the currently active process, message transfer requests from application-specific hardware, the presence of messages from other processors in the multiprocessor network, the presence of system errors, and the presence of hardware interrupts.

All events in the X32Carrier system that cause the current thread of instructions to stop and switch to another thread, another process, another code object, are processed as messages, using almost identical information structures for their processing. The only exception is switching to another process by timer, it is handled by the context controller on its own. Messages can be triggered both by a software request - the SENDMSG instruction, and by hardware ones. Hardware messages are interrupts and violations of the protection system.

Messenger arbitrates incoming requests and selects the highest priority ones. Messenger makes the necessary checks to see if the message initiator can access the message handler. Messenger processes requests, generates a set of parameters for the context controller, and calls the context controller to perform the procedure for switching the core to the desired process and the desired entry point to the message handler.



The messenger is also controlled by a firmware controller equipped with the appropriate firmware. This firmware controller is the same hardware as the context controller, but the firmware is different and the environment hardware is different.

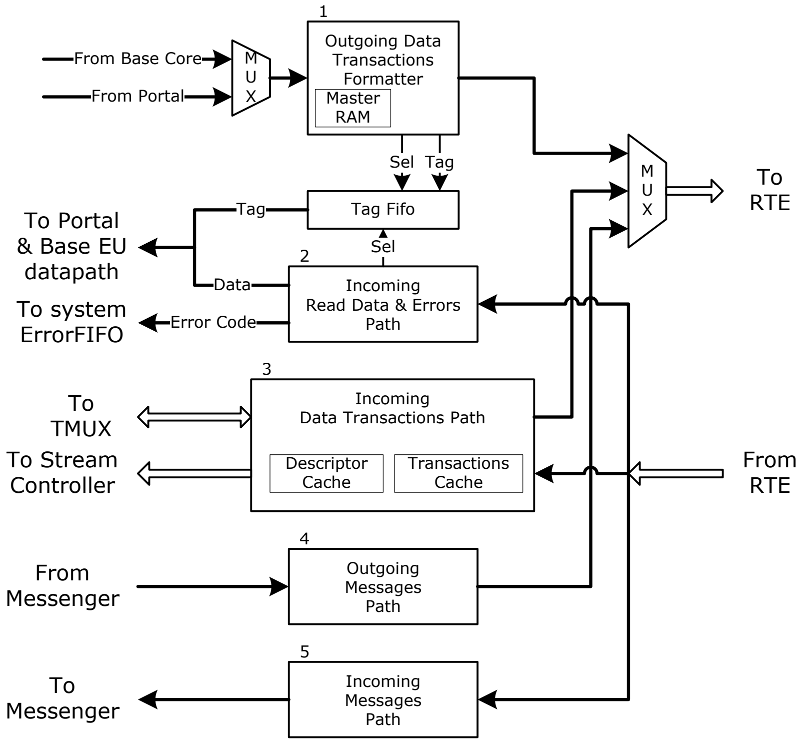
The four input ports of the microcontroller are fed through additional multiplexers with global processor parameters /Number, table limits, interrupt table selector, etc./, message parameters from the base execution unit, application-specific resource portal, message parameters, etc. received from other cores of the multiprocessor network, as well as the parameters of the current process.

Interface registers are used to pass prepared information to the context controller for switching to another process and to pass a message to another core of the multiprocessor network via the FPU.

The control register is used to generate various control signals and to control data multiplexers at the inputs of the microcontroller ports.

Frame Processing Unit – FPU.

The FPU performs multiprocessor transaction framing upon requests from the core and an application-specific resource and receives transaction frames from the multiprocessor network addressed to its own core.



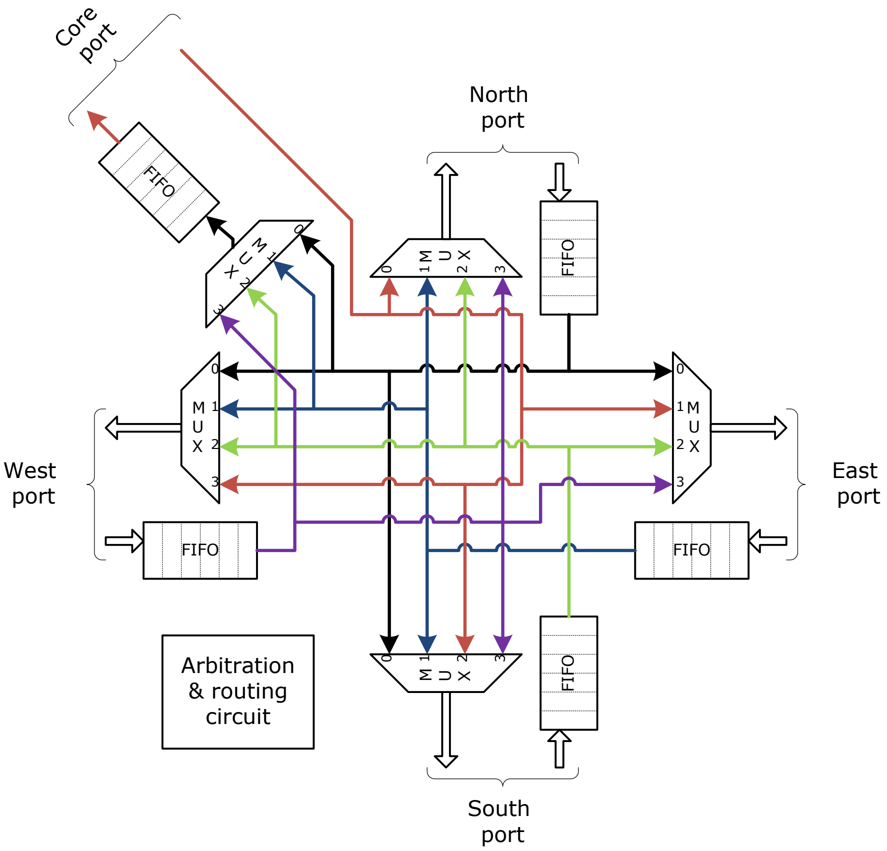
The FPU has five information channels.

1. Channel for outgoing data transactions. Each read or write transaction is stored in MasterRAM for subsequent determination of the multiprocessor frame generation mode in long or short form. Also, read transaction tags are placed in a special multi-channel queue, so that they can then be retrieved from there when data arrives from another core.

2. The returned data channel. It accepts responses to read transactions, as well as packets of system errors. The read data is tagged from the queue and sent to the underlying execution unit or portal. Error bursts cause the corresponding codes to be sent to the system error queue. And if an error was caused by a data read operation, then empty data is returned to the BEU or Portal, accompanied by the "not a number" format. This is done in order to close an open read transaction in the BEU or ASR.
3. Channel of incoming transactions. Processes requests for reading and writing data from other cores of the multiprocessor network. Contains 2 buffers - descriptor cache and transaction cache. The last cache is used to support the mechanism for processing short data frames. The channel generates read/write requests to TMUX and write requests to the flow controller. Having received the read data from TMUX, the channel generates a special frame to return to the processor that requested it.
4. Outgoing messages channel. Packs the messages prepared in Messenger into message frames and sends them to the multiprocessor network.
5. Incoming messages channel.

Routing Engine – RTE.

The engine is used to route frames of multiprocessor transactions. The engine has 5 ports: a local core port and four external ports north, east, south and west.



On the transmitting side of each port there is a 4-input multiplexer for transmitting outgoing frames from four other ports, and on the receiving side there is a queue of incoming frames.

The arbitration and routing circuit decides from which port queue and where the next frame will be sent.

Application-specific resource.

The architecture of the X32Carrier processor is adapted to support a specialized data processing resource by a system service, which can have its own specialized set of registers and a set of specific machine instructions.

Two types of dedicated hardware resource are allowed:

- a resource used by different processes in time-sharing mode;
- a resource used by various processes in a mode as it is ready to perform a new task.

The resource used by processes in time-sharing mode is characterized by the fact that each time the processor switches to another process, it is forced to stop and the context of a certain set of registers is stored in the PSO of the suspended process and loaded from the PSO of the newly activated process, if necessary. A resource used by timesharing processes can be temporarily idle if the currently active process is not using it. At the same time, it may not even receive a new register context due to the lack of one in the process.

The resource used by the processes when they are ready to accept a new job does not have a register context stored in the PSO processes. A resource of this type can perform the intended operation even after the process that initiated the operation is temporarily or permanently stopped.

An application-specific X32Carrier resource can contain both of these types. Part of the resource can be used by different processes in time-sharing mode, and part in ready-to-use mode.

An application-specific resource connects to the X32Carrier core through a portal. The composition and purpose of the tires of the portal are presented in the table below.

Bus	***Width Direction***	***Description***
	Instruction subsection	
PortalReady	1 ASR→Portal	The readiness of the application-specific resource to accept the next instruction. 0 - the resource is not ready to execute a new instruction.
PortalActivation	1 Portal→ASR	Activation of a new instruction for ASR. 1 - indicates the presence of a new instruction on the PortalInstruction bus.
PortalInstruction	32 Portal→ASR	ASR instruction code. The instruction code is translated from the instruction stream to the ASR without modification.

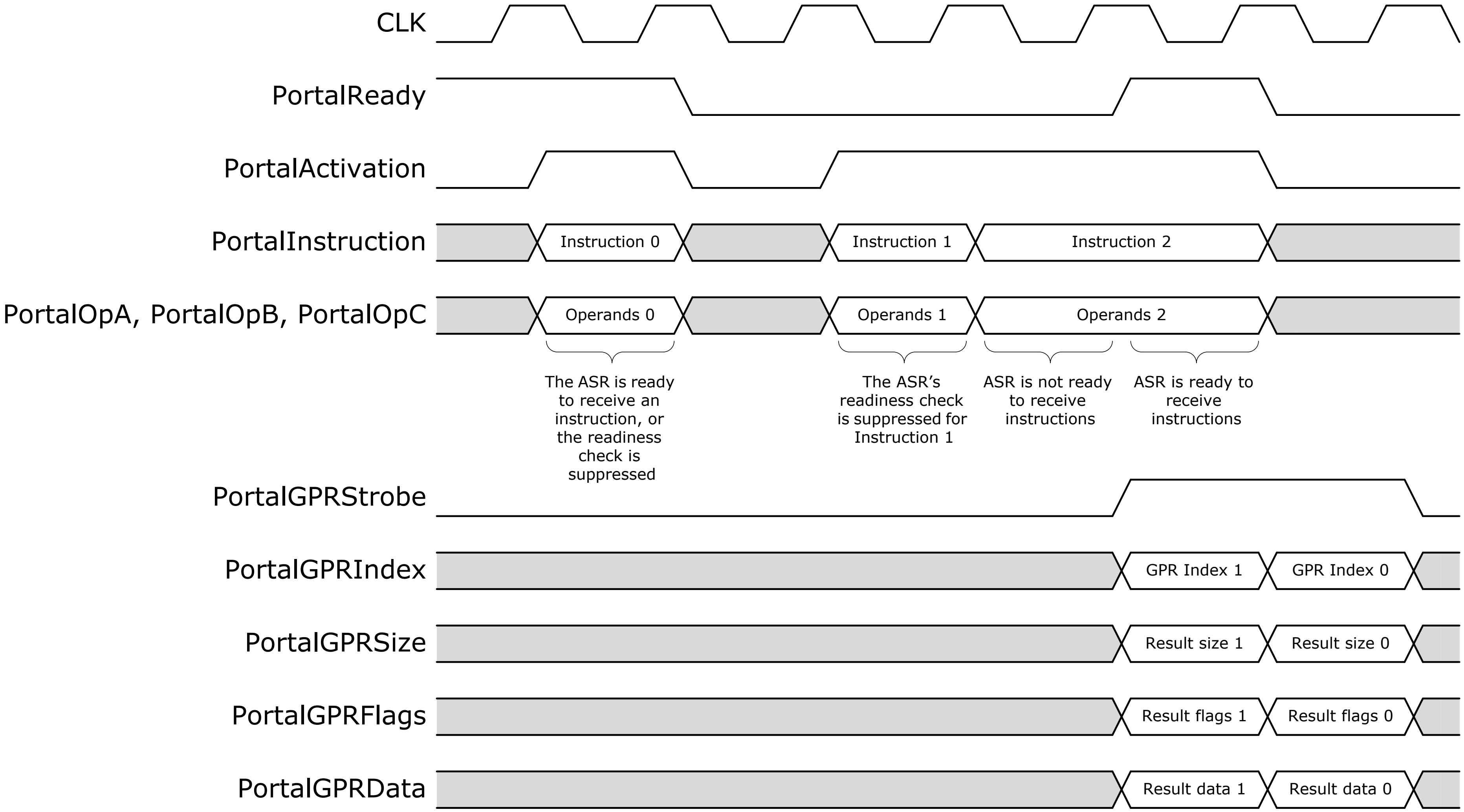
Bus	***Width Direction***	***Description***
PortalOpA PortalOpB PortalOpC	128 Portal→ASR	Source operands retrieved from the base unit's general purpose registers. Three 128-bit values are always extracted from the registers addressed in the ASR instruction by the fields [12:8] for PortalOpA, [20:16] for PortalOpB and [28:24] for PortalOpC.
PortalGPRStrobe	1 ASR→Portal	Result strobe in a general purpose register. With PortalGPRStrobe=1, data from the PortalGPRData bus is entered into a general-purpose register addressed by the PortalGPRIndex bus.
PortalGPRIndex	5 ASR→Portal	The index of the general purpose register that receives the result of the operation from the ASR. Usually this index is extracted from bits [28:24] of the Porta-lInstruction, since these are the bits that are commonly used to indicate the destination of the result.
PortalGPRSize	3 ASR→Portal	The width of the operand placed in the general register.
PortalGPRFlags	7 ASR→Portal	Operation result flags to be entered in the AFR flags register corresponding to the PortalGPRIndex index. PortalGPRFlags[0]→CF15 PortalGPRFlags[1]→ZF PortalGPRFlags[2]→SF PortalGPRFlags[3]→OF PortalGPRFlags[4]→IF PortalGPRFlags[5]→NF PortalGPRFlags[6]→DBF
PortalGPRData	128 ASR→Portal	The result of the ASR operation to be written to the selected general register.
Context load/store subsection		
PortalRST	1 Portal→ASR	System reset. Raised if there is a system reset and if there is a system error caused by ASR and if it is allowed to stop the resource in the presence of an error. Active level - 0.
PortalSTOP	1 Portal→ASR	ASR stop signal. Active 1. Formed by the context controller when it is necessary to stop the execution of the current process in the ASR and change its context to execute another process.
PortalRUN	1 Portal→ASR	ASR start signal after stop. Active 1. Raised by the context controller after the process context is loaded into the resource.

Bus	***Width Direction***	***Description***
PortalEMPTY	1 ASR→Portal	Sign of the absence of operations in ASR. Active 1. On receipt of PortalSTOP=1, the ASR shall terminate all its operations and set the PortalEMPTY=1 flag, allowing the context controller to perform a context save of the ASR registers.
PortalContextSTB	1 Portal→ASR	Context word strobe in ASR. With PortalContextSTB=1, the ASR must write to its PortalContextAddress-indexed register the 64-bit word set on the PortalContextDI bus.
PortalContextAddress	8 Portal→ASR	ASR register address. When the context is unloaded from the ASR, the context controller sets the address of the register in the bus, and on the next cycle, the word from the PortalContextDO bus is received for writing to the PSO of the process. 1 cycle is allocated for the selection of a word in the ASR. When the context is loaded, the register address, data, and strobe signal are set at the same time.
PortalContextDI	64 Portal→ASR	Context data to load into the ASR registers.
PortalContextDO	64 ASR→Portal	Context data returned from the ASR when the context is downloaded to the PSO.
Messages subsection		
PortalSMSG	1 ASR→Portal	Request to send a message. ASR generates PortalSMSG=1 when sending a message to messenger. The signal shall be held at 1 until an acknowledgment of receipt of the message is received.
PortalMSG	64 ASR→Portal	Message code. Index for the table of imported messages in bits [31:0], message parameter in bits [63:32]. The message code must also be kept valid until an acknowledgment of receipt of the message is received.
PortalSMSGNext	1 Portal→ASR	Message receipt confirmation. PortalSMSGNext=1 indicates that the message has been received by messenger in the current clock.
Memory interface subsection		
PortalNEXT	1 Portal→ASR	Readiness of the portal to accept a transaction from ASR in the current cycle. PortalNEXT=1 indicates that the transaction activated by ASR will be accepted for processing by the portal.
PortalERROR	1	System error flag. Set to 1 when the portal de-

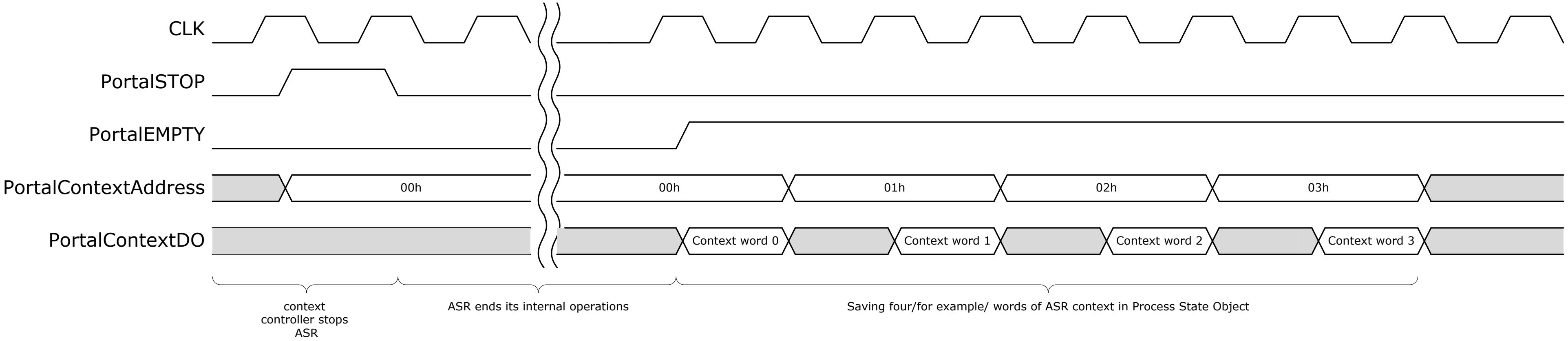
Bus	***Width*** ***Direction***	***Description***
	Portal→ASR	tects a memory access violation by ASR. The flag is advisory in nature and can be used by the ASR to make its own decisions regarding an erroneous transaction, the tag of which is indicated on the PortalERRORTag bus.
PortalERRORTag	8 Portal→ASR	The tag of the transaction that caused the memory access error. Valid for PortalERROR=1.
PortalACT	1 ASR→Portal	Transaction activation. PortalACT=1 The ASR is requesting a read or write transaction.
PortalCMD	1 ASR→Portal	Transaction type. PortalCMD=0 – write transaction, PortalCMD=1 – read.
PortalSelector	32 ASR→Portal	The selector of the object being accessed.
PortalOffset	37 ASR→Portal	The offset of the data in the object.
PortalDO	64 ASR→Portal	In transactions, writes are used to transfer the data being written.
PortalTO	8 ASR→Portal	Transaction tag. Used to identify the source and destination of data within the ASR.
PortalSO	2 ASR→Portal	Operand size: 8, 16, 32 or 64 bits.
PortalDRDY	1 Portal→ASR	read data readiness. PortalDRDY=1 indicates that data has been read from memory.
PortalDI	64 Portal→ASR	Data read from memory.
PortalTI	8 Portal→ASR	The tag of the transaction that previously requested the data to be read.
PortalSI	2 Portal→ASR	The size of the read operand.

8-bit data is transferred on the PortalDO and PortalDI buses on lines [7:0], 16-bit data on lines [15:0], 32-bit data on lines [31:0], and 64 bits use the full width of the buses .

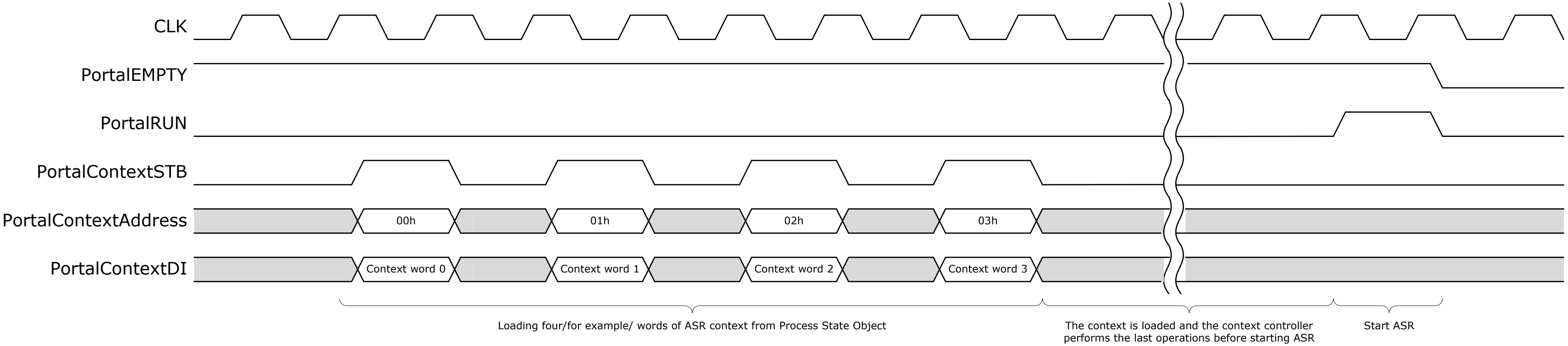
Instruction subsection diagram.



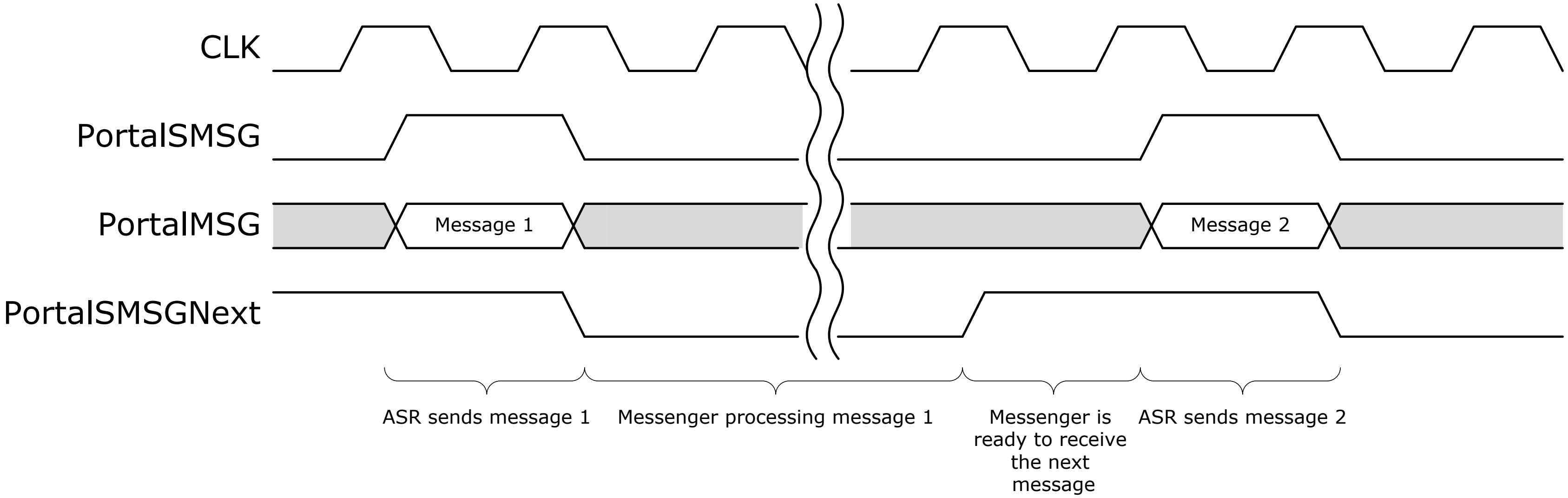
Context store diagram.



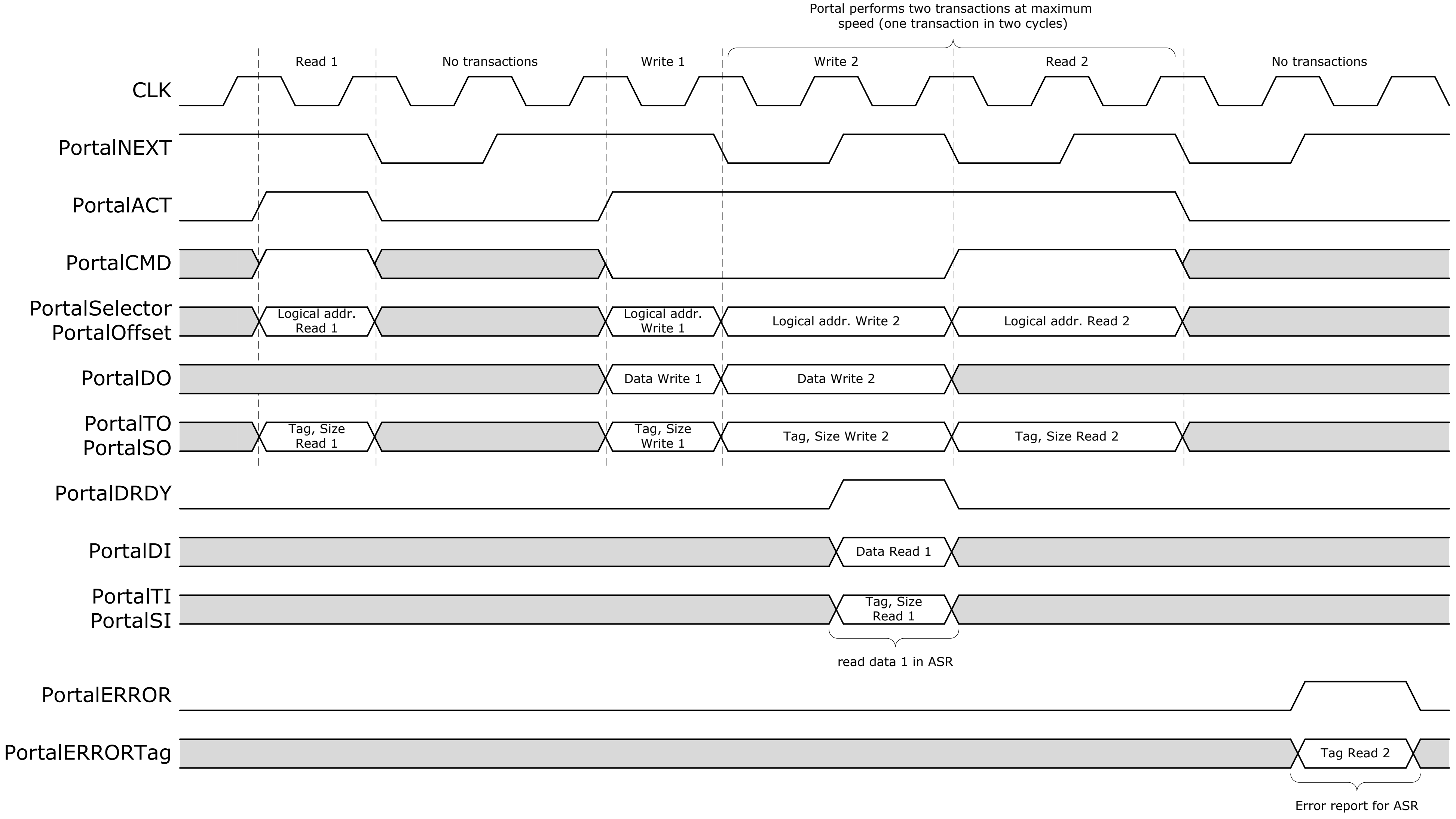
Context load diagram.


CLK
PortalEMPTY
PortalRUN
PortalContextSTB
PortalContextAddress
00h
01h
02h
03h
PortalContextDI
Context word 0
Context word 1
Context word 2
Context word 3
Loading four/for example/ words of ASR context from Process State Object
The context is loaded and the context controller performs the last operations before starting ASR
Start ASR

Messages subsection diagram.



Memory interface subsection diagram.



System registers.

System registers are mapped to memory space and assembled into a 128-byte block starting at physical address 1FFFFFFFFF80h. These registers are accessed by the system software through an object with selector 003h. Access to this object is possible only at the 0th privilege level. Prior to the initialization of the address translation system, access is possible through a pair of address registers AR10,AR11 which corresponds to the virtual register MAR5. Descriptor register DTR5 is initialized after a system reset to address memory starting at address 1FFFFFFFFF80h. The remaining registers DTR[4:0] and DTR[7:6] are initialized to base address 0. For more information about the composition of the system control registers, see the corresponding document: "X32 and X32Carrier Registers.pdf"