Dynamically Reconfigurable Resource Array

High Level Architecture Specification

Document Information

Nature	Technical report
Number	DRRA-June 0.5
Description	High Level Architecture Specification of DRRA
Access Level	Confidential (only under NDA)
Authors	Arun Jabalayan, Hasan Sohofi, Ahmed Hemani

Revision History

Comment	Version Number	Date
Initial Version	0.5	15/06/2016

Contents

1 Introduction

Dynamically Reconfigurable Resource Array (DRRA) architecture is developed to target physical (PHY) layer of the seven communication layers of the Open Systems Interconnection model (OSI). The DRRA technology can be instantiated as stand alone macros in a System on Chip (SoC) or as a part of a wireless system. To enable the feature of instantiating multiple parallel partitions, DRRA is designed as a Coarse Grain Reconfigurable Computational Fabric. The following are the three levels of granularity matching, supported by DRRA which is required by any computer architecture.

- Bit-Width Matching
- Instruction Matching
- Silicon Granularity Matching

Granularity matching allows the composition of an arbitrary partition consisting of FSM/Datapath to match the granularity of the algorithm. We call this partition a Coarse Grain Instruction(CGI).

Bit-Width Matching: DRRA is designed to target DSP applications and for most DSP applications the data width is 16 bits. The DRRA data width is a generic parameter which can be changed at any point in time during the design cycle by simply changing the value of the parameter.

Instruction Matching: To create a CGI, the distributed DRRA cells can be instantiated and configured in the proper mode such that the granularity matches the algorithm. Instruction granularity improves performance and energy efficiency due to reduced interconnect and memory operations.

Silicon Granularity Matching: Hardware structures like branch predictors, cache memory which are not directly involved in the computation might lead to silicon granularity mismatch and usually consume more energy. Such hardware structures mainly help in improving the execution speed. Usually, signal processing applications do not require random memory accesses or random jumps, and hence the DRRA does not require a branch predictor. Cache memories are replaced in the architecture with the help of a distributed memory architecture (DiMArch). Furthermore, each DRRA cell has an individual sequencer which helps in reducing the hardware complexity, as discussed later.

Granularity Matching The coarse granularity implies complex instructions and is fundamental to the higher performance of CGRAs because in a single Coarse- Granular-Instruction more computation is done.

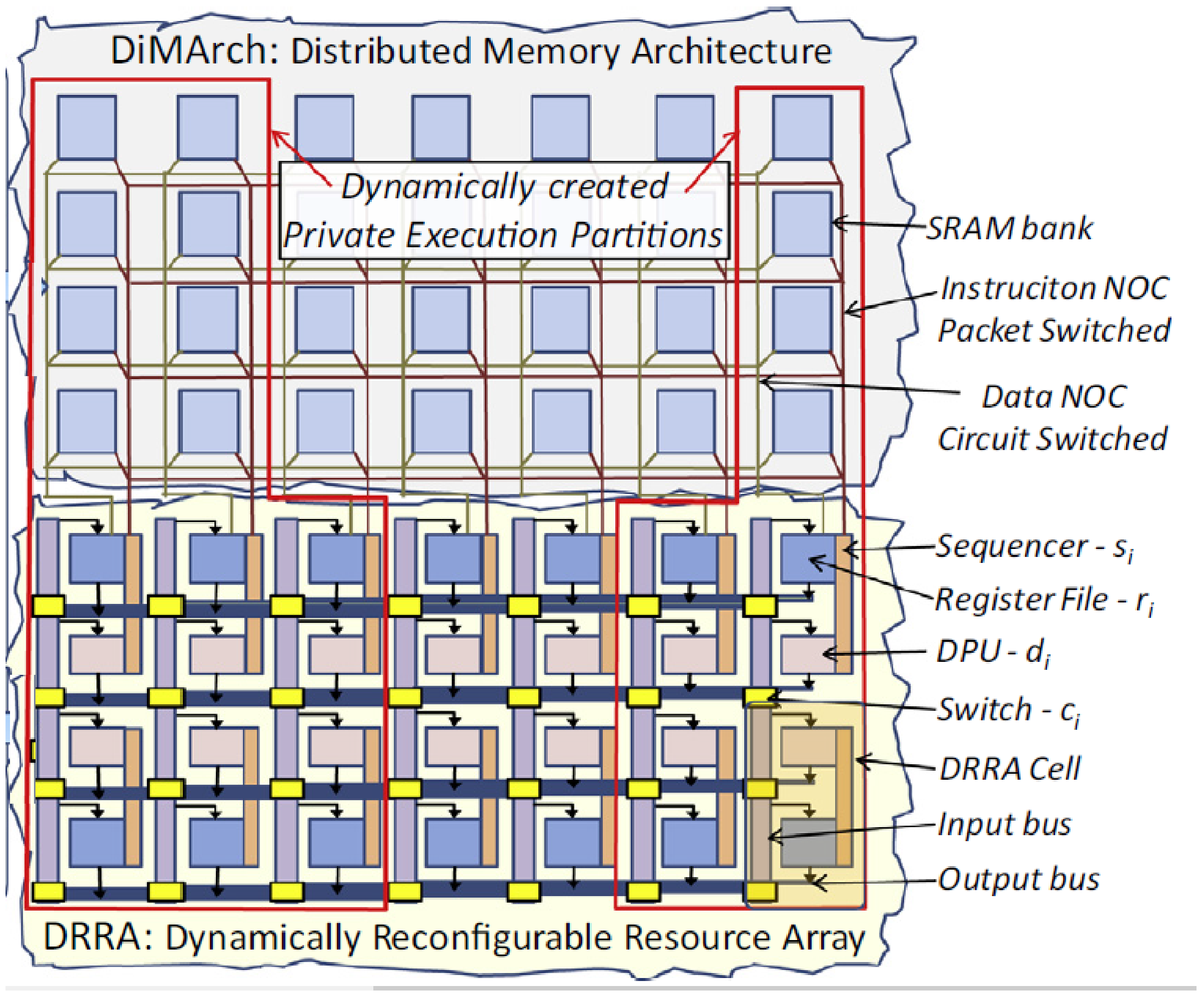

DiMArch: Distributed Memory Architecture
Dynamically created Private Execution Partitions
SRAM bank
Instruciton NOC Packet Switched
Data NOC Circuit Switched
Sequencer - s_i
Register File - r_i
DPU - d_i
Switch - c_i
DRRA Cell
Input bus
Output bus
DRRA: Dynamically Reconfigurable Resource Array

Figure 1: Dynamically Reconfigurable Resource Array

Figure 1 shows the diagram of silego fabric which has the DRRA cells and Distributed Memory Architecture. The number of rows and columns are set to 2 and 5 are design time decision and it can be changed during the design cycle. Each DRRA cell is connected to other cells through interconnects, using a sliding window 2-hop communication scheme. Adoption of 2-hop communication scheme is a design decision, considering a trade-off between clock speed, wiring overhead & the DPU width and also most of the signal processing data path cases, eight DPU and RFiles are often enough. A DRRA cell consists of a register file (RFile), Data Path Unit (DPU), Sequencer and switchboxes, each of these components are explained in the following sections.

2 Data Path Unit

Computational and logical resources of the DRRA fabric is collectively called the Data Path Unit(DPU). The DPU is partitioned into arithmetic, logical and pre & post-processing units shown in figure 2. Arithmetic operations are taken care by the arithmetic partition and it provides standard signal processing operations like FFT butterfly modes, symmetric and asymmetric MAC (Multiply and Accumulate) modes. The Arithmetic partition supports both integer and fixed point operations. Logical AND/OR, comparison, arithmetic shift, logical shift operations are performed by the logical partition. Counter operations are carried out by a dedicated timer/counter unit which is part of the DPU. The last partition provides pre and post processing functions like negation, saturation, truncation etc.

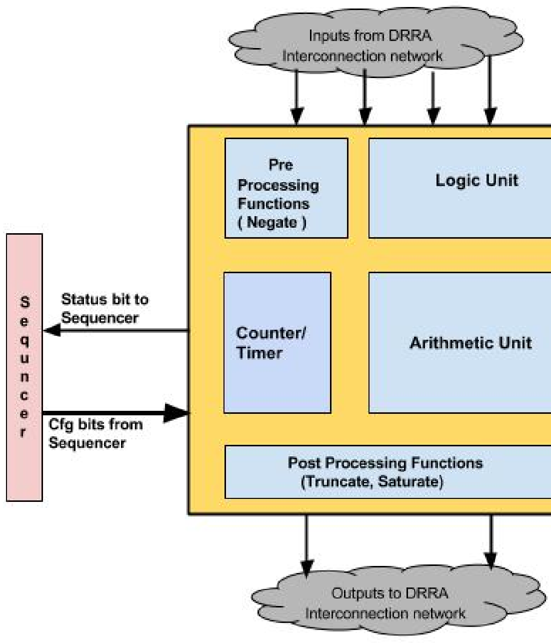

Inputs from DRRA Interconnection network
Pre Processing Functions (Negate)
Logic Unit
Sequencer
Status bit to Sequencer
Counter/ Timer
Arithmetic Unit
Cfg bits from Sequencer
Post Processing Functions (Truncate, Saturate)
Outputs to DRRA Interconnection network

Figure 2: An illustration of a Data-Path Unit used in the DRRA fabric

Input output signals of the DPU is shown in Figure 3. The DPU has four 16-bit signed input and signed output ports. Different DPU operations can be selected by configuring a 5-bit signal named dpu_mode_cfg. Currently the DPU supports 12 modes of operation and it is discussed in Table 3.

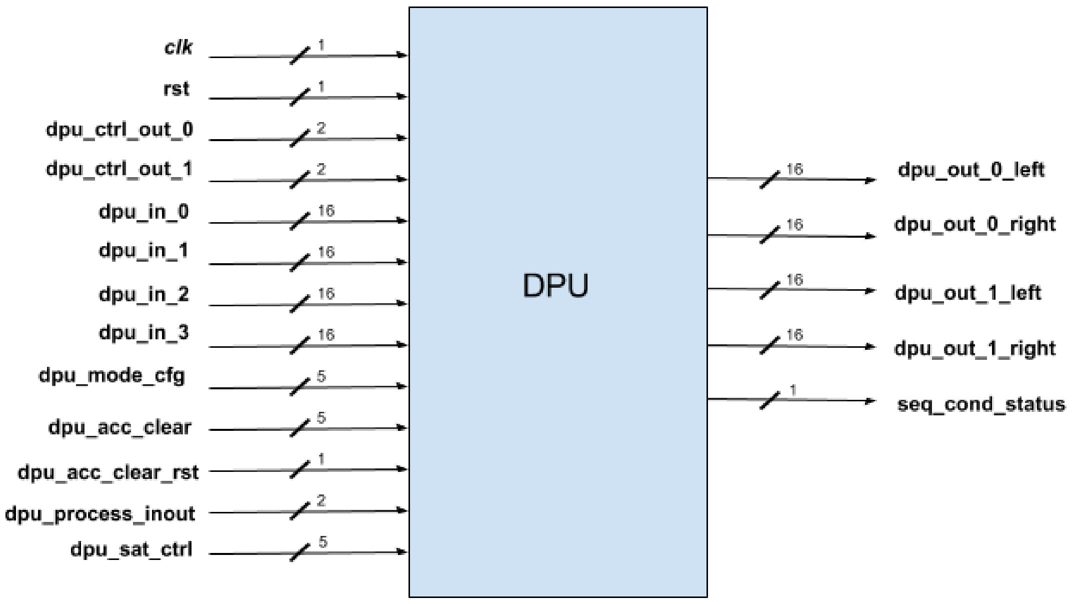


Figure 3: Data Path Unit in/out Ports

The DPU has two 2-bit control signals $dpu\_ctrl\_out\_[0:1]$ to enable or disable the output ports. It also has two dedicated signals, $dpu\_acc\_clear\_rst$ and dpu_acc_clear to control the accumulator register. The $dpu\_acc\_clear\_rst$ is a 1-bit asynchronous active high reset signal to clear the accumulator register while the 5-bit dpu_acc_clear control signal clears the accumulator register when the accumulator counter reaches the value configured in dpu_acc_clear. Pre-processing and post-processing operations are controlled by another 5-bit control signal dpu_sat_ctrl and a 2-bit $dpu\_process\_inout$. Pre and post processing operations can be performed by programming appropriate values as listed in the Table 2. An output signal seq_cond_status acts as a control signal to the local sequencer while executing branch instructions. Since the seq_cond_status is only used by the sequencer to take a decision, it is available at the outport in the same clock cycle when the inputs are available at inport. The DPU has two 16-bit output ports $dpu\_out\_[0:1]\_[left, right]$, the left & right ports route the output to right and left directions

The MAC operation includes arithmetic functions like multiplication, convolution (such as FIR, IIR and correlation), transformations (like FFT and DCT) and also normal arithmetic operations. Input data to the DPU is supplied by the register files of any DRRA cell provided it is within the sliding window.

The DPU supports both unsigned and signed data formats, and also integer and fixed point operations. The DPU uses Q-15 fixed point format which has 1-bit to represent a sign bit and 15-bits to represent the fractional part.

The DPU operations are pipelined in nature therefore, arithmetic operations excluding multiply take a single clock cycle, while multiply operations take two clock cycles to complete the execution.

The preprocessing unit performs negation and absolute operations while the post processing unit performs operations like truncation etc. The input and output to the DPU is a 16-bit number.

However, the result of a multiplication is 32 bits, which after accumulation becomes 33 bits. Since output ports are 16-bit wide, the result should be truncated before it is produced at the output. The post processing partition with the help of functions like truncation, saturation and rounding converts arithmetic result with more than 16 bits into a 16-bit number.

The DPU is controlled by a local sequencer of the DRRA cell. Some of the functions include selecting the DPU modes, setting the constants to which the results are compared, preprocessing the inputs, clearing accumulator register after N counts and postprocessing functionality. The sequencer also receives inputs from DPU status bits and decides the appropriate action to be taken.

2.1 DPU Signals

The list of DPU signals is shown in Table1. All control signals are active high except reset.

Table 1: DPU Signals

Signal Name	**Width**	**Description**
clk	1	Synchronous clock for the DPU unit
$reset$	1	DPU Reset
dpu_mode_cfg	5	Configures the DPU modes between 1 and 12
$dpu\_in\_[0:3]$	16	input ports
$dpu\_out\_[0:1]\_[left, right]$	16	output ports
$dpu\_ctrl\_out\_[0:1]$	2	Selects left or right output port
$dpu\_acc\_clear\_rst$	1	clears the accumulator register
dpu_acc_clear	5	clears the accumulator register when accumulated value matches the value programmed in dpu_acc_clear
dpu_sat_ctrl	5	selects out data format
$seq\_cond\_status\_width$	2	control signal to the sequencer

Table 2 explains the control signals and its modes of operations.

Table 2: DPU control signals and its mode of operation

Signal Name	**Type**	**Values**	**Description**
		0	disables out ports
		1	Out port 0 will be enabled
$dpu\_ctrl\_out\_[0:1]$	Control	2	Out port 1 will be enabled
		3	Out port 0 & 1 will be enabled
		0	Integer operation & No saturation
		1	Fixed point & No saturation
dpu_sat_ctrl	Control	2	Integer with saturation
		3	Fixed point with saturation
		0	No preprocessing
		1	Negates input 0
$dpu\_process\_inout$	Control	2	Negates input 1
		3	Absolute of (in0-in1)

We consider the following notations:

- $x_k[i]$, $y_k, left[i]$, $y_k, right[i]$ the input operands and the output results of the i^{th} sample.
- $k \in \{0..3\}$ for x and $k \in \{0, 1\}$ for y. left and right are the direction of the output results.
- acc and mul are two internal registers to store intermediate accumulation and multiplication results.
- $y_k[i]$ and $y_k[i]$ are two intermediate output registers storing the results before routing to either right or left directions.

Mode	Operation	Description
$Mode\_0$	IDLE	$y_0[i] = 0$ $y_1[i] = 0$
$Mode\_1$	MAC	$acc = y_0[i-1]$ $mul = (x_0[i] + x_1[i]) * x_2[i]$ $\downarrow$ $y_0[i] = mul + acc$ $y_1[i] = mul$
$Mode\_2$	MUL	$acc = y_0[i-1]$ $mul = x_0[i] * x_2[i]$ $dpu\_cnt\_clear = 1$ $\downarrow$ $y_0[i] = mul + acc$ $y_1[i] = mul$
$Mode\_3$	MAC	$acc = x_3[i-1]$ $mul = x_0[i] * x_2[i]$ $\downarrow$ $y_0[i] = mul + acc$ $y_1[i] = mul$
$Mode\_4$	Radix 2 FFT Mode	$acc = x_3[i]$ $mul = x_0[i] * x_2[i]$ $\downarrow$ $y_0[i] = mul + x_3[i]$ $y_1[i] = x_1[i] - mul$
$Mode\_5$	Radix 2 FFT Mode	$acc = 0$ $mul = x_0[i-1] * x_2[i]$ $\downarrow$ $y_0[i] = mul + x_3[i]$ $y_1[i] = x_1[i] - mul$
$Mode\_6$	Min Max Mode	$y_0[i] = max(x_0[i], max\_temp\_reg)$ $y_1[i] = min(x_0[i], min\_temp\_reg)$ $acc\_clr\_cnt\_en = 1$
$Mode\_7$	Absoulute Mode	$y_0[i] = abs(x_0[i] - x_1[i]) + acc$ $y_1[i] = 0$ $acc\_clr\_cnt\_en = 1$
$Mode\_8$	COUNTER Mode	value stored in dpu_in_0 controls counter operation $up\_count = 0x7$

		$down\_count = 0\text{x}4$ $set\_count = 0\text{x}5$ $reset\_count = 0\text{x}6$ $stop\_count = 0\text{x}8$
$Mode\_9$	Arithmetic Shift Mode	$y_0[i] = (x_0[i] << x_1[i])$ $y_1[i] = (x_0[i] >> x_1[i])$
$Mode\_10$	ADD/SUB	$y_0[i] = x_2[i] + x_3[i]$ $y_1[i] = x_0[i] - x_1[i]$
$Mode\_11$	COMP	$y_0[i] = 0\text{XFFFFFFFF}\ if\ in_0 > in_1$ $y_1[i] = 0\text{XFFFFFFFF}\ if\ in_2 > in_3$
$Mode\_12$	Load Constant & Add Constant	$y_0[i] = const$ $y_1[i] = x_0[i] + const$

Table 3: DPU modes of Operations

2.2 DFG and Timing Diagram

The data flow graph(DFG) of the DPU is shown below figure 4

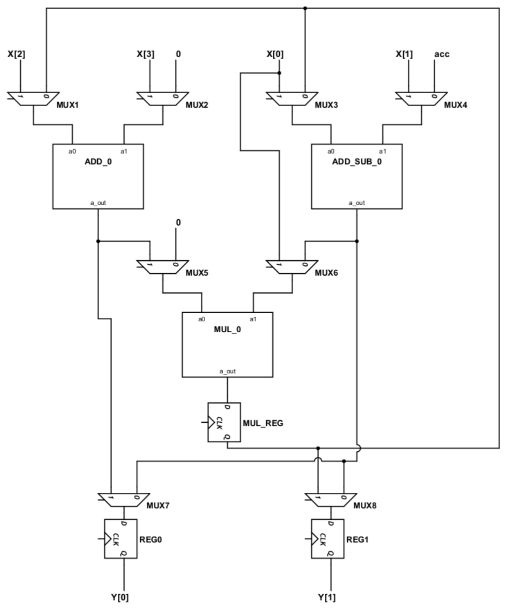


Figure 4: Data Flow Graph of the DPU

Figure 5 shows the timing diagram for Arithmetic and logical operations in vector mode. These operations takes single clock cycle to calculate the result, hence output is delayed by one clock cycle.

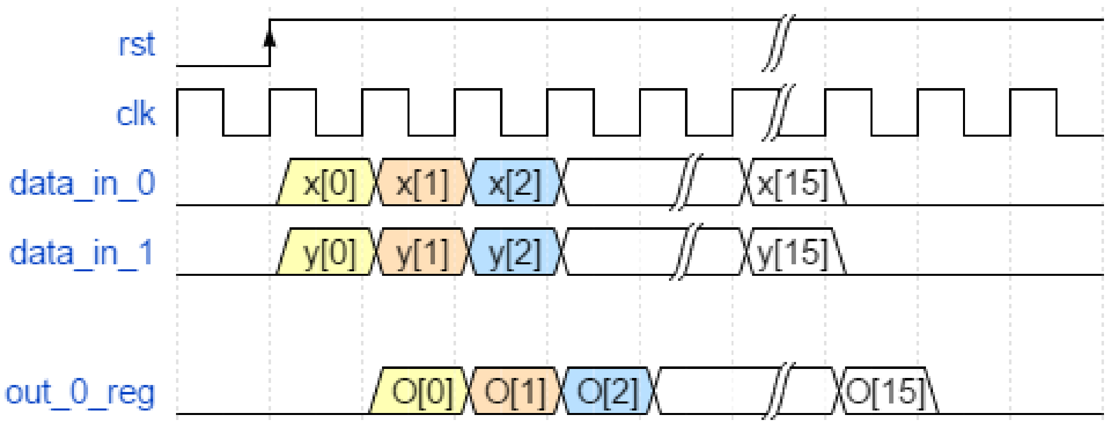


Figure 5: The DPU timing diagram for Arithmetic(no multiplication) & logical operations

The timing diagram for MAC operations in vector mode is shown in Figure 6. MAC operations takes two clock cycles, one clock cycle to compute multiplication and one clock cycle for the addition, hence output is delayed by two clock cycle.

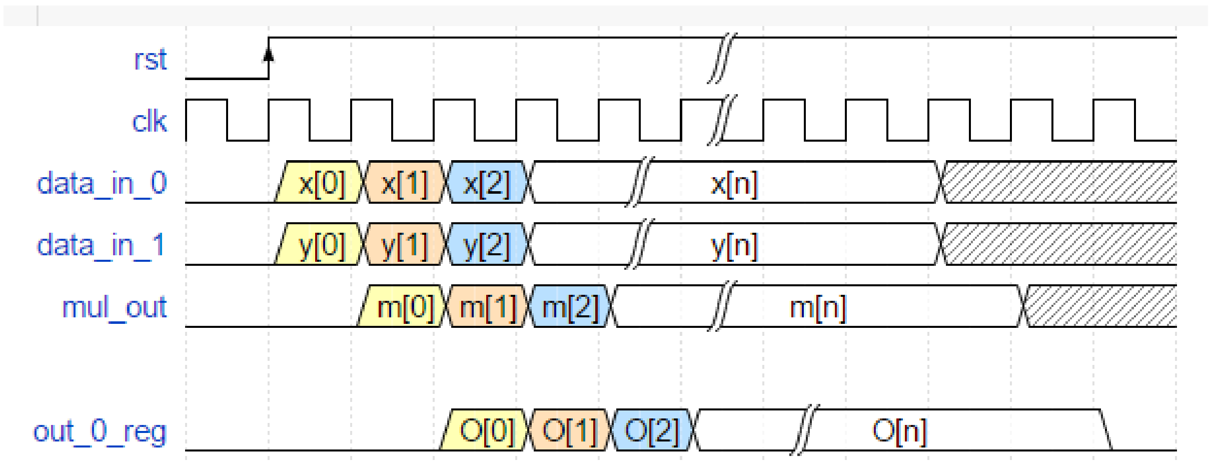


Figure 6: The DPU timing diagram for MAC operations

3 Register File

In the DRRA, the RFile serves the purpose of providing high bandwidth parallel data to the DPUs thereby achieving the local high bandwidth and distributed storage for implementing the Coarse Grained Instruction (CGI). All the data computed by the DPU is sourced and sunk to the RFiles and/or DPU of other DRRA cells. The RFile has two read and two write ports and one bidirectional port to memory interface. The memory port of width 256 bits, is dedicated for data transfer between the memory and Register file. Each read and write port has a dedicated address generation unit (AGU) which provides the read and write addresses to the RFile. The depth of the Registerfile is 32 and each of those registers can store 16-bits of data. Similar to the DPU, the depth and width of the RFile can also be changed in the design cycle.

The data movement between the RFile and the DPU in the DRRA cell happens in the same clock cycle. The data movement between the register bank and memory, takes (2+N) clock cycles for the WRITE operation while 2+2N clock cycles for the READ operation, where the 2 clock cycles are required to configure the switch(router) while 'N' is equal to the number of Hops.

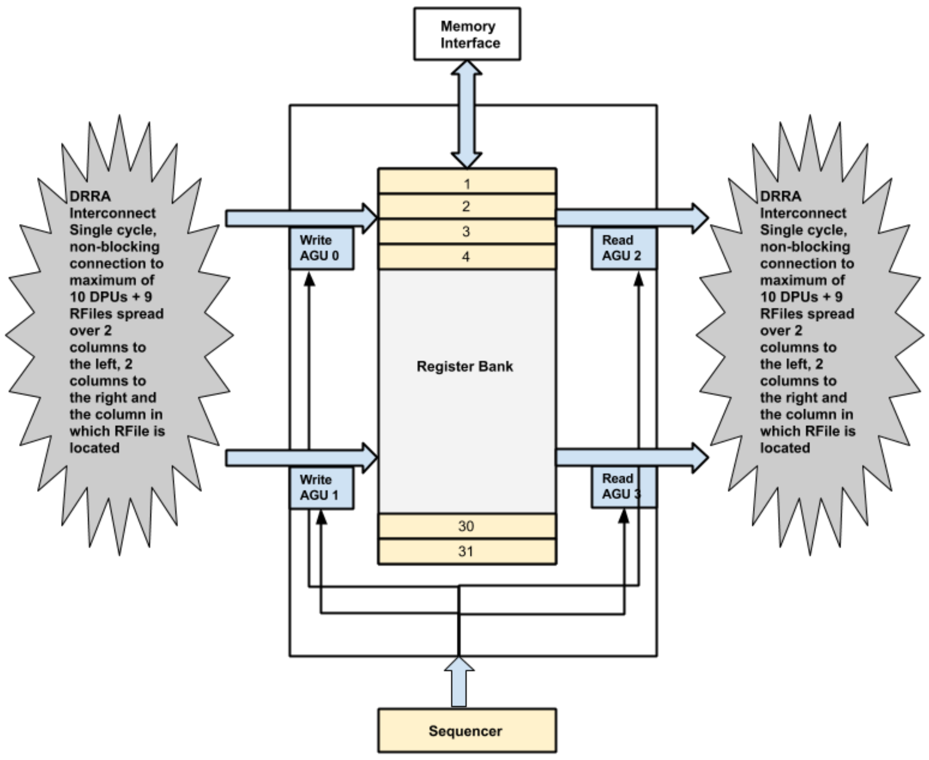

Memory Interface
DRRA Interconnect Single cycle, non-blocking connection to maximum of 10 DPUs + 9 RFiles spread over 2 columns to the left, 2 columns to the right and the column in which RFile is located
Write AGU 0
1
2
3
4
Register Bank
30
31
Read AGU 2
Write AGU 1
Read AGU 3
DRRA Interconnect Single cycle, non-blocking connection to maximum of 10 DPUs + 9 RFiles spread over 2 columns to the left, 2 columns to the right and the column in which RFile is located
Sequencer

Figure 7: The RFile Block Diagram

Programming Interface The programming interface shown in Figure 7 receives the AGU configuration instructions. The instruction contains the AGU port addresses, the programming interface routes the instructions to the appropriate AGU.

In and out signals of the RFile is shown in figure 8. The RFile has a control signal $dimarch\_mode$ which controls the data flow; when this signal is assigned the value 1, data transfer happens between the memory and RFile. The $rd\_inst\_start\_0$ and $wr\_inst\_start\_0$ control signals help to identify the memory operation as a read or write respectively. The $wr\_addr\_[0:1]$ and $rd\_addr\_[0:1]$ signals hold the write and read addresses respectively while $wr\_data\_ready\_[0:1]$ and $rd\_[0:1]$ control signal indicates if the WR and RD address are valid or not. Input signal $data\_in\_[0:1]$ holds the data to be written into the RFile from the DPU while the $dimarch\_data\_in$ signal holds the memory data to be written in the RFile. Signals $data\_out\_reg\_[0:1]\_[right, left]$ and $data\_out\_2$ holds the output data to the DPU and Memory respectively.

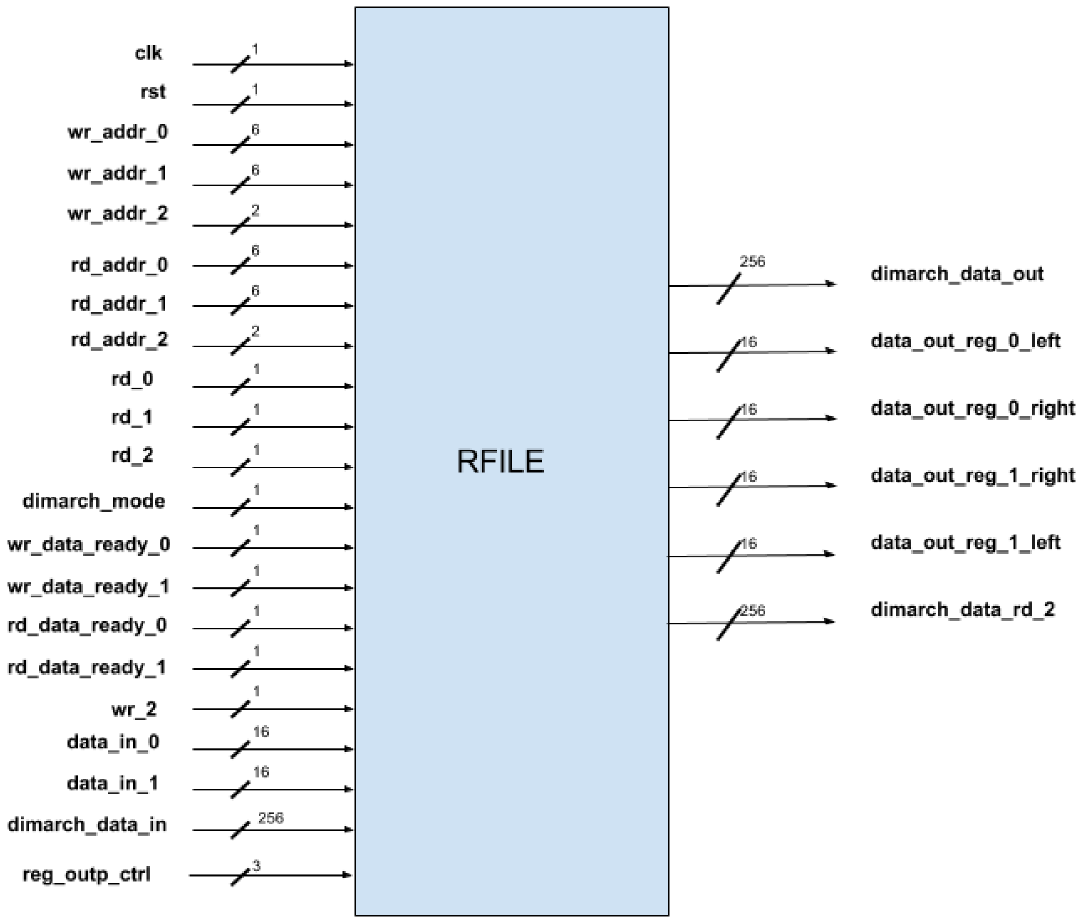


Figure 8: RFile in/out Ports

3.1 RFile Signals

The list of DPU signals is shown in Table4. All control signals are active high except reset.

Table 4: RFile Signals

Signal Name	**Width**	**Description**
clk	1	clock for the RFile unit
$reset$	1	RFile Reset
$dimarch\_mode$	1	If set, Rd/Wr happens between Memory and RFile
$rd\_inst\_start\_0$	1	Decides the mode of operation(Rd, Wr, Both)
$wr\_inst\_start\_0$	1	in conjunction with dimarch_mode
$wr\_addr\_[0:1]$	6	RFile Write address
wr_addr_tb	2	RFile WR address (Memory Block)
$wr\_data\_ready\_[0,1,tb]$	1	If 1 then WR data is valid
$rd\_addr\_[0:1]$	6	RFile RD address
rd_addr_tb	2	RFile RD address (Memory Block)
$rd\_[0:1]$	1	If 1 then read data is valid
$dimarch\_rd\_2\_out$	1	If 1 then mem data is valid
$data\_in\_[0:1]$	16	Write data to RFile
$data\_in\_2$	256	Write memory data to RFile
reg_outp_cntrl	2	Selects out port
$data\_out\_reg\_[0:1]\_[left, right]$	16	Output ports
$data\_out\_2$	256	Out data to write in Memory

3.2 Timing diagram

The timing diagram of the RFile for the read operation is shown in the figure 9. As observed from the waveform, the read address $rd\_addr\_[0:1]$ is valid only when the $rd\_[0:1]$ is valid. Likewise, the write address $wr\_addr\_[0:1]$ and the data $data\_in\_[0:1]$ are valid only if the $wr\_data\_ready\_[0:1]$ is active high.

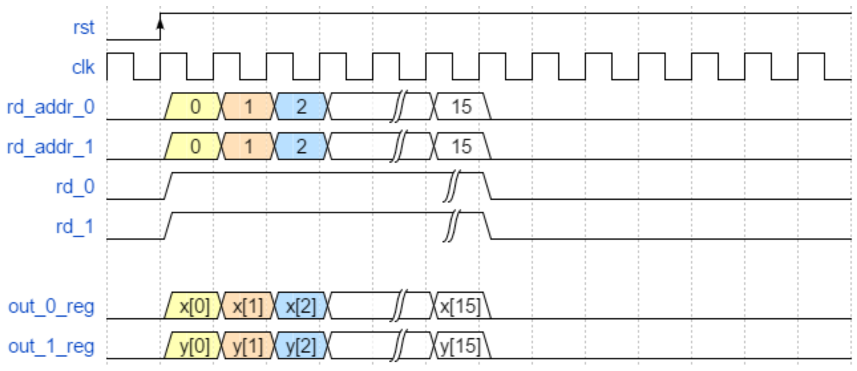
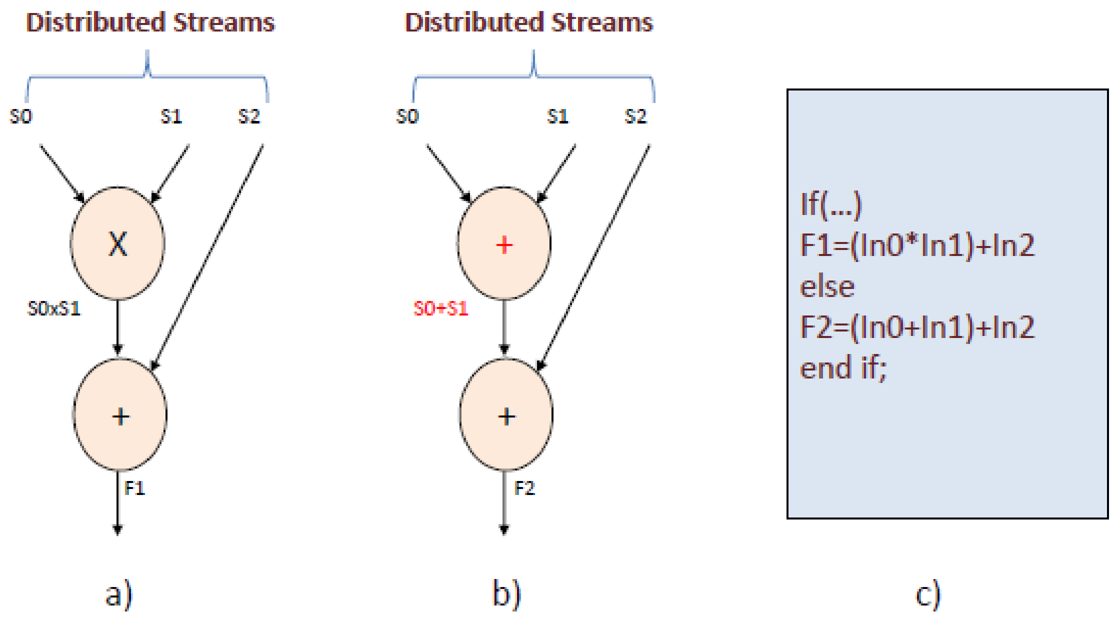


Figure 9: The RFile timing diagram for Read operation

4

Address Generation Unit

In general purpose processors (GPP), the addresses are random in nature and memory load/store instructions are partly used to compute the address. Hence ideally, an AGU has to generate a random addressing pattern. However, the memory access pattern in most of the DSP algorithms is a function of the previous address (Xi = f(Xi-1)), which means that the addresses are streaming in nature. Stream in this context is, a set of in and out data which has a periodic delay between consecutive data. So, the data access patterns will be generated by an address generation logic (AGU). This dedicated AGU logic reduces the number of instructions as it decouples the address generation logic from the computational unit. It also enables parallelism between the compute and memory operations. This is the reason DSP processors use a dedicated AGU for algorithms like FIR, and FFT etc.



```
If(...)
F1=(In0*In1)+In2
else
F2=(In0+In1)+In2
end if;
```

Figure 10: Streaming needs of an AGU in a PDDSP architecture

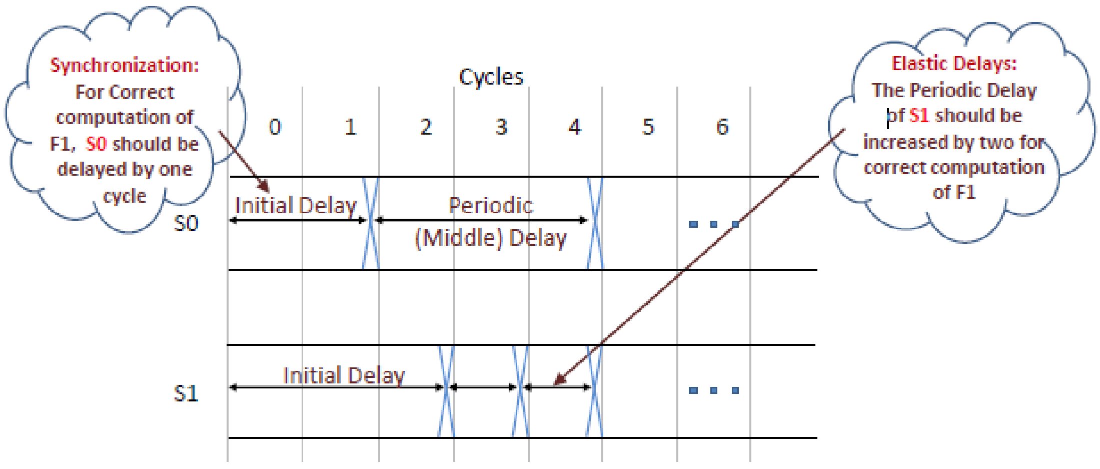

Synchronization: For Correct computation of F1, S0 should be delayed by one cycle
Cycles
0
1
2
3
4
5
6
Elastic Delays: The Periodic Delay of S1 should be increased by two for correct computation of F1
S0
Initial Delay
Periodic (Middle) Delay
S1
Initial Delay

Figure 11: An illustration to explain the needs for stream synchronization

The AGUs used in DSP processors can not be used in distributed computing architectures like DRRA as they have limited temporal and spatial properties. To exploit parallelism in distributed architectures, an AGU should have additional logic other than just calculating the addresses at regular instants. The requirements of an AGU for distributed architectures are mentioned below:

Dedicated Address Generation Modes Like any other AGU implemented for a DSP processor, the AGU for parallel distributed digital signal processing (PDDSP) architectures requires a dedicated addressing mode for target DSP applications like FFT, FIR, matrix arithmetic, vector read/write etc.

Ability to Generate Arbitrary Address The ability to generate arbitrary addresses provides flexibility to run any arbitrary algorithm and increase the overall flexibility of the system.

Repetition of an Instruction An AGU can repeat an instruction once, some finite number of times or infinitely unless interrupted. This prevents re-issuing of an instruction for the same addressing pattern and helps in reduction of code as well as code movement to the AGU.

AGUs Local to Distributed Storage The storage resources in a PDDSP are distributed and may provide data to one computational resource at one instance and to another at any other instance. These distributed storage resources need local AGUs to decouple the address generation logic from the compute logic and hence can be connected to any other computational resource at any instance in time.

Synchronization of Streams In a PDDSP architecture, a number of simultaneous read/write operations take place, in distributed storage resource, at regular intervals creating Streams. These streams are the inputs(outputs) to(from) the memory/compute resources. One or more compute resources computing an algorithm can consume two or more streams depending on the algorithm. In such cases there is a need to synchronize these streams for correct computational results There are two kinds of synchronization needed for the streams:

* **Arrival Time Synchronization**: In figure 10(a), to correctly compute the function F1, the streams S0 and S1 should arrive synchronous to each other while the stream S2 should arrive synchronous to the computation of S0 S1. Figure 11 shows that S0 arrives earlier than S1; therefore, correct computation of F1 is not possible.

* **Period Delay Matching**: Two streams can only be connected together if their periodic delay is the same. In order to connect two streams with different periodic delays, the delay of one of the streams will have to be changed to make it equal to the delay of the other stream. In Figure 10(a) and (b), to compute F1 correctly, the periodic delays of S0, S1 and S2 should be programmed equal to each other. Figure 11 shows that the period delay of S0 is 2 while the period delay of S1 is 0. Therefore, S0 and S1 cannot be connected without adjusting their period delay.

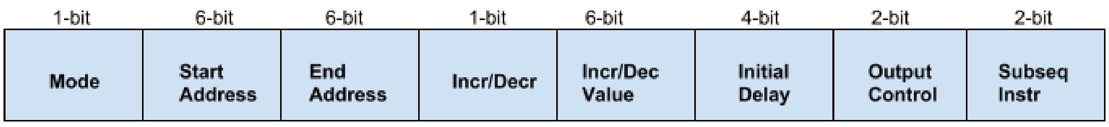
Elastic Stream A stream whose periodic delay can be changed at runtime is called an Elastic Stream. These streams can be connected to other streams with different periodic delays at runtime by configuring their elastic periodic delays. The elastic periodic delays are also needed to adjust branch specific delays. In Figure 10(c), one of the two possible data-paths will be selected based on the branch selected in the C Code. When the data-path implemented in Figure 4.10(a) is selected, the periodic delay of the S2 will be equal to the delay of the multiplier. However, if the branch shown in Figure 10(b) is selected, the periodic delay for S2 will be equal to the delay of addition. With an elastic stream, branch-specific delays of the stream can be adjusted at runtime.

The first three needs discussed above can be applied to an AGU in any kind of computer architecture and are discussed below. However, the last three needs are specific to PDDSP architectures like DRRA. In the following text we will describe the address generation mechanism in DRRA and will further discuss how these mechanisms are used to fulfill the needs discussed above. The addresses in DRRA are generated by the use of a small AGU internal to RFile that can generate commonly used address generation sequences. The AGU can work either on a 27 bits basic instruction, or a 54 bits extended instruction. The basic instruction defines the modes in which AGU operates to generate address to meet the requirements regarding address generation discussed above. The extended instruction sets the synchronization delays, data processing rate and number of times the instruction will be repeated to address the synchronization needs discussed above. The basic and extended instruction controls the spatial addressing and temporal behavior of address generation patterns.

The AGU supports two types of instructions namely basic and extended instructions which is discussed in the following sections.

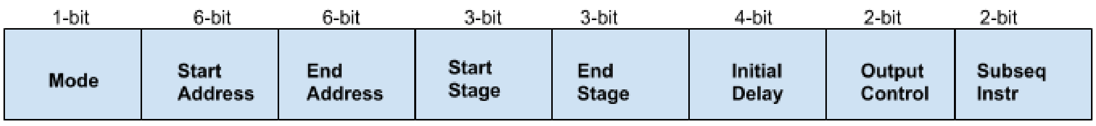
4.1 Basic Instruction

A basic instruction configures an instruction mode, start & end addresses and an initial delay while an extended instruction configures the middle delay, repeat delay and repeat values. The address generation sequence of the AGU is configured by the basic instruction. The AGU can generate two address modes namely Linear addressing and bit reversed addressing mode. The addresses for most DSP applications like FFT, FIR filters, delay lines, matrix multiplications can be generated using these two modes. The following two figures show the various fields comprising a 27-bit instruction and the width of each of those fields for both the addressing modes respectively.



1-bit	6-bit	6-bit	1-bit	6-bit	4-bit	2-bit	2-bit
Mode	Start Address	End Address	Incr/Decr	Incr/Dec Value	Initial Delay	Output Control	Subseq Instr

Figure 12: AGU Basic Instruction Vector Mode



1-bit	6-bit	6-bit	3-bit	3-bit	4-bit	2-bit	2-bit
Mode	Start Address	End Address	Start Stage	End Stage	Initial Delay	Output Control	Subseq Instr

Figure 13: AGU Basic Instruction Bit Reversed Mode

The $subsequent\ instruction$ field in the basic instruction differentiates the basic from the extended instruction. When the $subsequent\ instruction$ bits are set to a non zero value, it is configured in the extended mode, hence the local sequencer considers the following two instructions in the instruction register and forms a complete instruction. The instruction mode determines the mode of operation, which when set to 1, prompts the AGU to generate a bit reversed address otherwise a linear address is generated.

Linear Addressing Mode: In the linear addressing mode, the AGU generates linear addresses for read and write ports of the RFile. The start and end addresses in the instruction forms the start and end address of the address generation pattern. The instruction configures the AGU, such that the next address can be incremented or decremented by a step value. It also configures an initial delay which is inserted at the instance before the address pattern is generated. This delay helps to achieve synchronization between the ports of the same RFile or distributed RFile.

Bit Reversed Mode: In the bit reversed mode, the AGU generates bit reversed addresses for FFT operation. As mentioned earlier, the bit reversed mode can be selected by setting the $instr\_mode$ to 1 . As a design decision, in the FFT mode, the start address should be either 0 or 1. As bit reversing depends on the FFT stage, the instruction is designed to generate addresses for one FFT stage or a range of FFT stage like 2-stage, 4-stage, 8-stage and so on, by configuring the start and end stage in the instruction. Currently, the AGU in the bit reversed mode generates FFT addressing upto 5 stages, as the depth of the RFile is 32. Hence we can perform up to 32 point FFT operation using a DRRA cell. Further, it is possible to perform FFT beyond 5-stages by combining two or more DRRA cells.

4.2 Extended instruction

It allows us to control the synchronization and timing of the basic instruction. For this purpose, in addition to the initial delay in the basic instruction, the extended instruction introduces two more delays which are a)Middle Delay and b)Repetition Delay. This results in an address generation pattern with configurable temporal behaviour and is shown in Figure 14. The addresses are generated using the basic instruction of the AGU. The delays and repetition patterns are set by using the extended instruction of the AGU. Note that the delays can be automatically synthesized by the compiler.

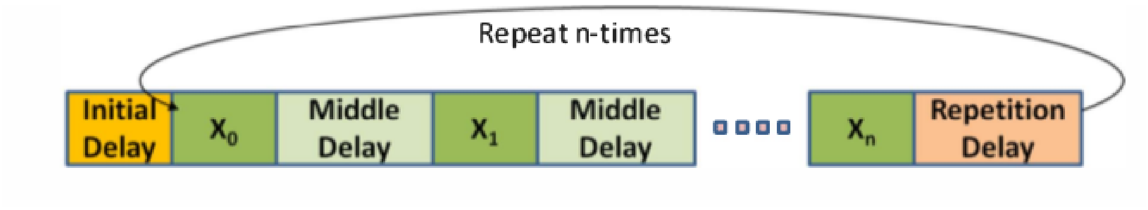


Figure 14: Address Generation Pattern with initial, middle and repetition delays to support synchronization in DRRA.

Initial Delay: The Initial delay is a programmable delay of maximum 16 clock cycles. Typically, it is used to delay the stream of data by a number of clock cycles to achieve synchronization between independent streams at the same time.

Middle Delay: It is possible to introduce a programmable delay between two consecutive addresses using the middle delay. In order to perform an operation between two streams of data, it should have the same periodic delay. The maximum middle delay possible is 16 clock cycles.

Repetition Delay: Before repeating the address generation cycle, it is possible to introduce a delay called the repetition delay, for instance in a scenario where the AGU is waiting for a new data to arrive after completing one cycle. Synchronization can be achieved by programming the repetition delay and the repetition of an address pattern can be delayed by a maximum of 64 clock cycles.

Repeating a Pattern: An address pattern can be repeated for a maximum of 32 times by configuring the $number\ of\ repeat$ field in the instruction. This avoids the reconfiguration of the AGU and reduces the code generation for the same instruction. It is possible to increment or decrement the start address by an offset value before repeating the address generation, which can be achieved by assigning the $repeat\ step\ value$ field a value between 0 and 15 in the instruction.

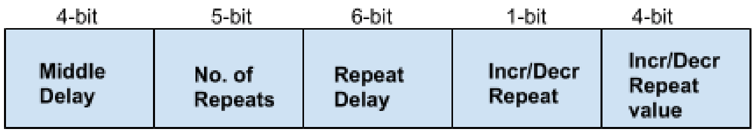


Figure 15: AGU Basic Instruction - Extended Instruction

4.3 AGU FSM

A finite state machine shown in the figure 16 controls the address generation as per the values programmed in the basic and extended instructions. The state machine has two address generation states namely $LINEAR\_ST$ and FFT_ST and three synchronization states namely $INITIAL\_DELAY\_ST$, RPT_DELAY_ST and RPT_ST. The FSM begins from the IDLE state. When the $instr\_start$ field in the instruction is set to 1, the state machine checks for the initial delay. If the initial delay is programmed, the state machine will go to the $INITIAL\_DELAY\_ST$ state, otherwise it will directly go to the $LINEAR\_ST$ or the FFT_ST state depending on the value of the $instr\_mode$ bit. If the middle delay field is assigned a particular value, the program counter will be halted until the middle delay counter reaches the middle delay value. If the repeat delay field is configured, the state machine will jump to the RPT_DELAY_ST state at the end of address generation cycle, otherwise the state machine will enter into the RPT_ST state and increments/decrements the starting address of the instruction.

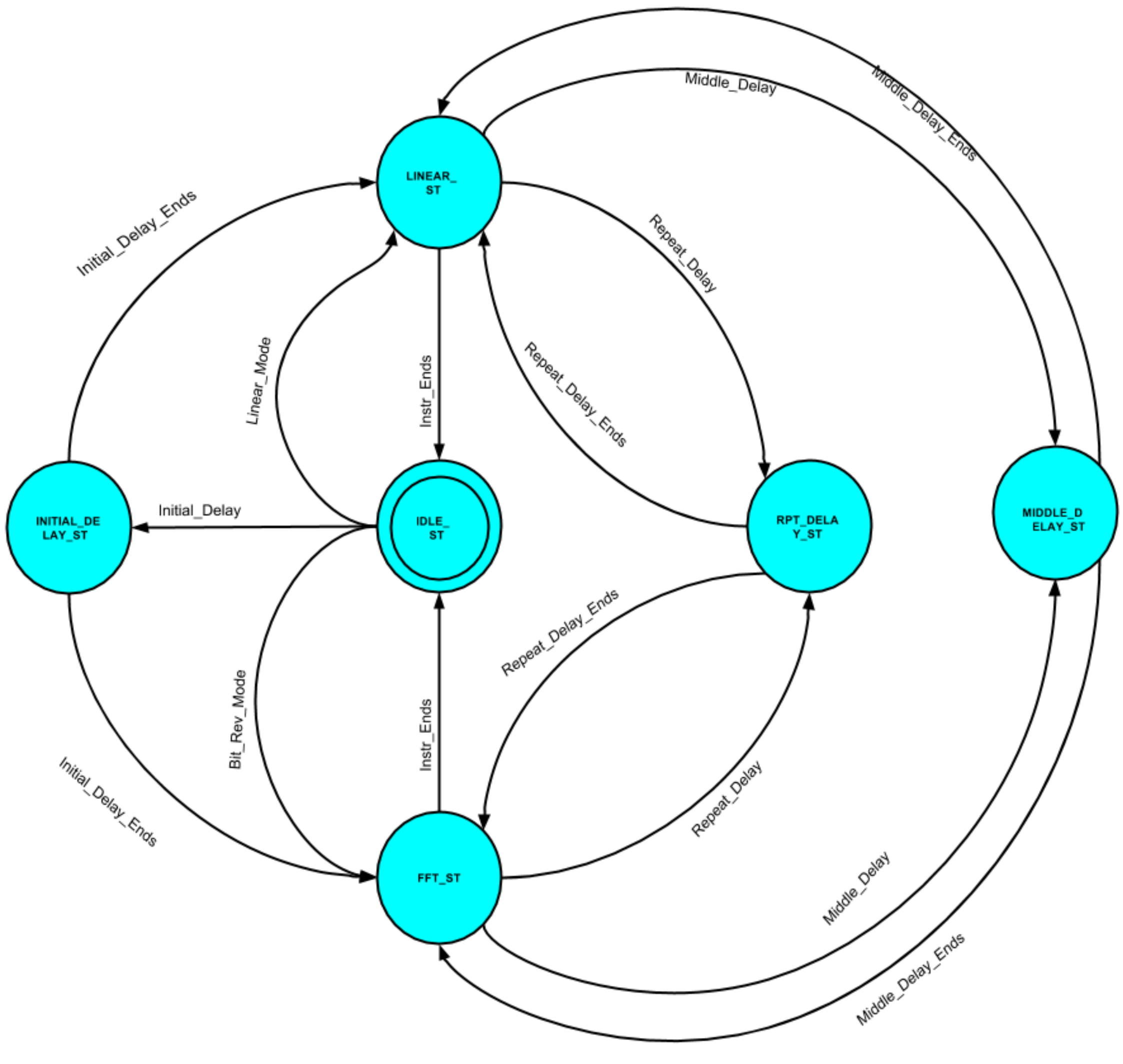


Figure 16: The working of AGU using its state diagram

4.4 List of Signals

Table 5: AGU Signals

Signal Name	**Width**	**Description**
clk	1	Synchronous clock to the AGU unit
$reset$	1	Asynchronous reset to the AGU
$instr\_start$	1	Instruction load begins only if $instr\_start$ is set
$instr\_initial\_delay$	4	Initial delay
$instr\_start\_addrs$	6	Starting address of the address pattern
$instr\_step\_val$	6	Difference between two consecutive addresses
$instr\_step\_val\_sign$	1	If 0, next address is incremented by the $instr\_step\_val$ else decremented
$instr\_no\_of\_addrs$	6	Total number of addresses to be generated
$instr\_middle\_delay$	4	Delay between two consecutive addresses
$instr\_no\_of\_rpts$	5	Number of times the address pattern to repeat
$instr\_rpt\_step\_value$	4	Increment/decrement the start address by this value before repeating the pattern
$instr\_mode$	1	If 0, generates linear addresses else bit reversed address
$instr\_fft\_stage$	3	FFT start stage
$instr\_end\_fft\_stage$	3	FFT end stage
$addr\_out$	6	address out
$addr\_en$	1	$addr\_out$ is valid only if the $addr\_en$ is set

4.5 Timing Diagram

The linear address generated by the AGU with out any delay is shown in the figure 17. The starting address, step value, repeat step value, number of repeat and number of addresses are configured to 3, 2, 10, 1 and 3 respectively. Hence the address pattern generated is 3, 5, 7, 9, 13, 15, 17, 19, 23, 25, 27, 29.

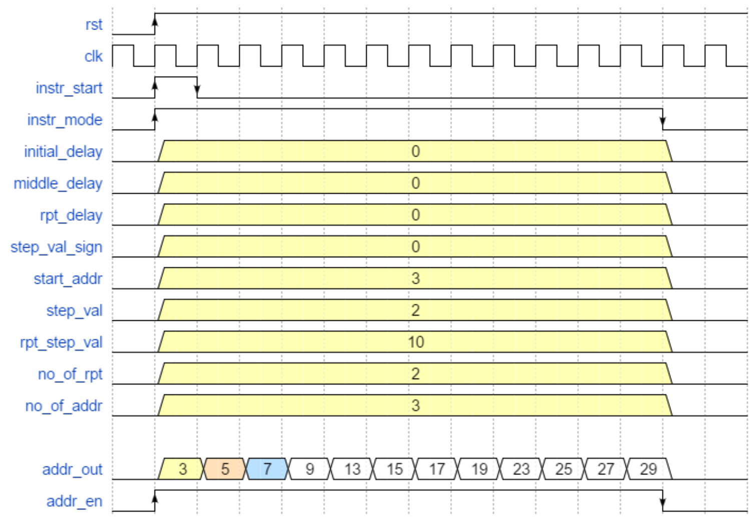


Figure 17: Linear Address Generation Pattern without any delay

Figure 18 shows the timing diagram for the above mentioned values with $initial\_delay = 1$, $middle\_dealy = 2$ and $repeat\_delay = 5$. Since there is an intermediate delay, the $addr_en$ is kept low during the delay period.

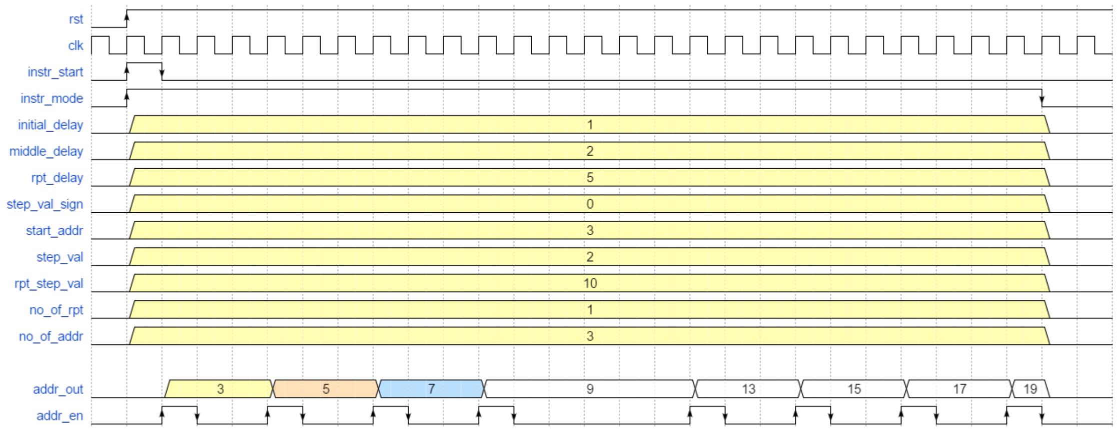


Figure 18: Linear Address Generation Pattern with delay

Figure 19 shows the timing diagram for the FFT mode with $initial\_delay = 1$, $middle\_delay = 1$, $repeat\_delay = 2$, $fft\_start\_stage = 1$ and $fft\_end\_stage = 5$. Bit reversed address will be generated for each FFT stage (i.e, from FFT stage 1, 2, .., 5). As the $no\_of\_rpt = 1$, the address pattern will be repeated once in each stage, however there will be a repeat delay of 2 clock cycles (as configured) before repeating the address pattern.

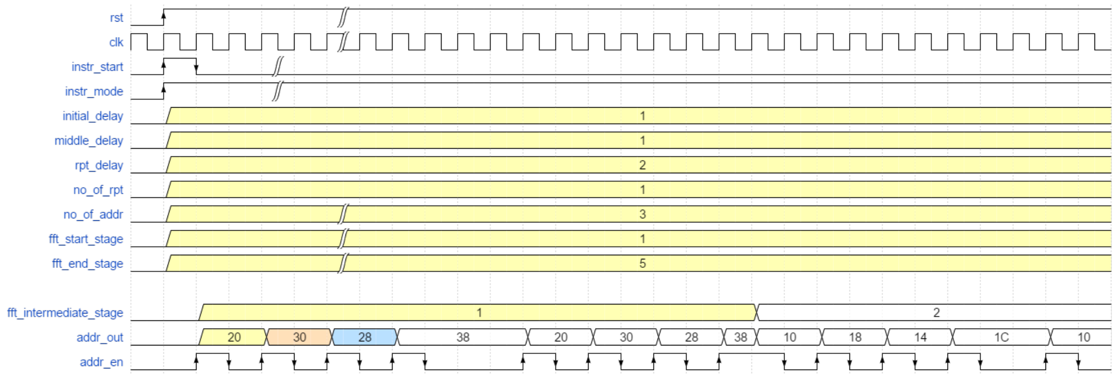


Figure 19: FFT Address Generation Pattern with delay

5 Sequencer

All resources in the DRRA cell are controlled by a micro-coded hierarchical state machine called the sequencer. Unlike the other general purpose processors architecture, each DRRA cell has a dedicated sequencer instead of having one sequencer which controls the chip. The main purpose of the sequencer is to configure the DPU, Switches and AGUs of RFiles dynamically. The sequencers also communicate with other sequencers within the sliding window, to maintain synchronization.

The key components of the sequencer are the program memory, and instruction decoder and a simple FSM that maintains the current program counter (PC). Besides sequential advancing of PC, unconditional and conditional jumps are also supported. The conditional jumps are used to implement while loops or IF-THEN-ELSE branching and are implemented using the status bits coming from the DPU telling the sequencer if some comparison condition has been met or not.

In DRRA, the sequencers have the following primary roles:

Creation of CGI The DPUs and RFiles provide the needed resources to create an arbitrary size CGI. However, it is the sequencer which configures them in appropriate mode and the interconnect network to connect them together. It is this ability of configuring DPUs and RFiles in an appropriate mode along with interconnect configuration that makes a sequencer capable of creating a partition consisting of data-path/FSM combined. In order to make a bigger partition, two or more sequencers will configure their own DRRA cell resources to connect them together and will work in synchronization with each other. Using this approach, an arbitrary sized CGI can be created. The DRRA fabric can instantiate many such CGIs in arbitrary parallel, pipelined or time-multiplexed ways. The parallelism can be at arithmetic level, algorithmic level or application level.

Parallel Dynamic and Partial Reconfiguration The possibility of reconfiguring DPU, RFile and switchboxes by the sequencer instead of having a separate reconfiguration unit results in efficient implementation of partial and dynamic reconfiguration. The partial or dynamic reconfiguration behaviors are very easily embedded inside the program to achieve at runtime. Either a complete resource or a part of it can be reconfigured by the sequencer. For instance, a switchbox can be reconfigured completely or just one input can be routed to a different output. This allows the sequencer to dynamically reconfigure the CGI created. This will be useful when CGIs are used in time multiplexed fashion. The reconfiguration for the whole fabric is done in parallel by all the sequencers with every sequencer responsible for configuring resources in its DRRA cell. One DRRA cell has to configure six interconnection networks corresponding to six inputs, four RFile(Register File) AGUs, and a DPU. Inputs takes a single cycle to configure, while every AGU and DPU takes two cycles to configure. All the DRRA cells are configured in parallel by their own local sequencers.

DRRA Hierarchical Control Besides the local control, the DRRA sequencer also implements hierarchical control. To do this, the DRRA control interconnect is used. The control interconnect, like the data interconnect connects each sequencer to all sequencers, two columns on each side along with the other sequencer in the same row.

Program Memory The sequencer has a program memory of 32 instructions of 27 bits each. The 4 MSBs are used for the sequencer instruction code while the remaining 23 bits are used to store the instruction operands. With 4 bits as sequencer instruction code, only sixteen sequencer instructions are possible, however currently the sequencer 14 instruction types which is discussed in table 6.

Instruction Decoder The instruction decoder decodes the instructions which are fetched from the memory. The instruction decoder outputs configurations for DPU, RFile, switchboxes. It also supports conditional/unconditional branch instructions and delay instructions. The register file (RFile) instruction shown here is a single instruction; however, it can carry many different configurations. Similarly, the DPU and switchbox instruction can carry many different configurations based on the modes in which these components will be configured. The sequencer also controls the register file delay dynamically by using the initial, middle and repetition delay offset instructions. When the current instruction is configured in the extended mode, the sequencer decodes the consecutive 1 or 2 instructions depending on the $subsequent\ instr$ in the AGU instruction. For example, when $subsequent\ instr$ is configured to '1', the sequencer fetches the current and next instruction from the instruction register while the PC will be incremented by 2. When $subsequent\ instr$ is configured to '2', the current and next 2 instructions will be fetched by the sequencer and PC will be incremented by 3.

Sl.No	Instruction	Description
1	DPU	DPU configuration instructions
2	$REFI1$	RFile basic instructions
3	$REFI2$	Dynamic Middle & Repeat Delay Control of RFile
4	$REFI3$	RFile extended instructions
5	$JUMP$	Unconditional Jump EQ, NEQ,
6	SWB	Switchbox configuration instructions
7	$ROUTE$	Route instructions for the DiMArch
8	$DELAY$	Dealy instructions
9	$SRAM\ Read$	SRAM Read instructions
10	$SRAM\ Write$	SRAM Write instructions
11	$RACCU$	RACCU configuration instructions
12	$HEADER$	Loop Header instructions
13	$TAIL$	Loop Tail instruction

Table 6: List of Instructions supported by the Sequencer

The Sequencer state machine shown in the figure 20 has $IDLE\_ST$, $SEQ\_LOADING\_ST$ and $INSTR\_DECODE\_ST$ states. The state machine begins from the $IDLE\_ST$ state. It goes to the $SEQ\_LOADING\_ST$ state when the $instr\_ld$ signal is flagged. In the $SEQ\_LOADING\_ST$ state, the finite state machine loads the instructions corresponding to its own DRRA cell into the instruction register. After loading all the instructions, the state machine moves to the $INSTR\_DECODE\_ST$ state where it decodes each instruction and distributes it to the corresponding unit. After decoding all the instructions, the state machine enters into the $IDLE\_ST$ state and remains in the same state until a new instruction load happens.

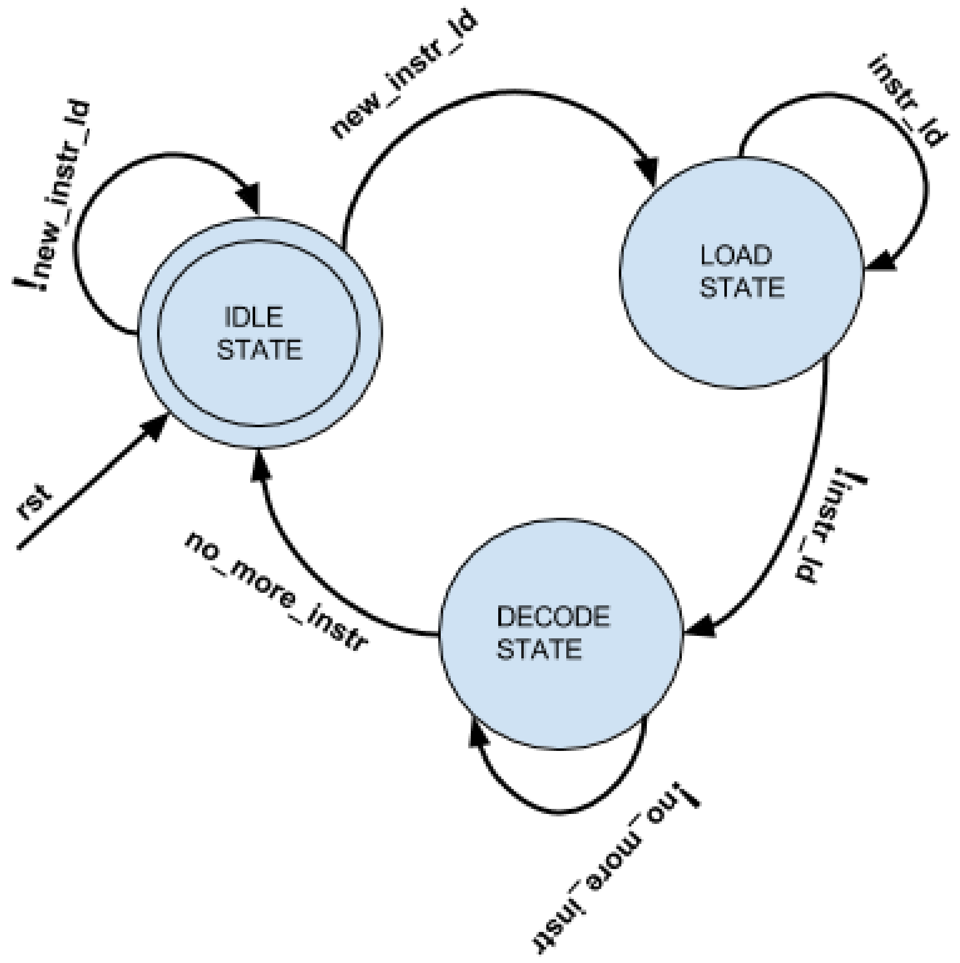


Figure 20: A state diagram governing the functionality of the Sequencer

5.1 DPU Instruction

DPU (dpu_mode, dpu_saturation, dpu_out_1, dpu_out_2,dpu_acc_clear_rst, dpu_acc_clear_sd,dpu_acc_clear, process_inout)

- *dpu_mode* - Configures the valid dpu mode (between 1 and 12)
- *dpu_saturation* - Selects the integer or fixed point operation with or without saturation as discussed in the Table 2
- *dpu_out*$[0:1]$ - Selects one or both the outports as discussed in the Table 2
- *dpu_acc_clear_rst* - Asynchronous reset, when set, clears the DPU accumulator register.
- *dpu_acc_clear* - The DPU accumulator register is cleared when the accumulator counter reaches the value configured in the *dpu_acc_clear*

- *dpu_acc_clear_sd* - The *dpu_acc_clear* is valid only when *dpu_acc_clear_sd* is set.
- *process_inout* - Processes the input or output as mentioned in the table 2

5.2 REFI_1 Instruction

REFI_1 (reg_file_port, subseq_instrs, start_addrs_sd, start_addrs, no_of_addrs_sd, no_of_addrs, initial_delay_sd ,initial_delay)

- *reg_file_port* - Selects one of the RFile ports 0, 1, 2, 3
- *subseq_instrs* - The instruction decoder fetches the consequent (REFI_1) or (REFI_1 and REFI_2) instructions.
- *start_addrs* - Configures the starting address for the AGU
- *start_addrs_sd* - The *start_addrs* is valid only if the *start_addrs_sd* is $'0'$. Otherwise the start address would be taken from the RACCU register
- *no_of_addrs* - Configures the number of addresses to be generated by the AGU.
- *no_of_addrs_sd* - The *no_of_addrs* is valid only when *no_of_addrs_sd* is $'0'$, otherwise the *no_of_addrs* would be taken from the RACCU register.
- *initial_delay* - Configures the initial dealy

5.3 REFI_2 Instruction

REFI_2(step_val_sd, step_val, step_val_sign, refi_middle_delay_sd , refi_middle_delay, no_of_reps_sd, no_of_reps, rpt_step_value)

- *step_val* - Step incremental/decremental value of the address
- *step_val_sd* - The *step_val* is valid only if *step_val_sd* is set.
- *step_val_sign* - If '0' address will be incremented by the *step_val* else decremented by the *step_val*
- *refi_middle_delay* - Configures the middle dealy
- *refi_middle_delay_sd* - The *refi_middle_delay* is valid only if *refi_middle_delay_sd* is $'0'$.
- *no_of_reps* - Configures the number of times the address pattern to repeat.
- *no_of_reps_sd* - The *no_of_reps* is valid only when the *no_of_reps_sd* is $'0'$ otherwise the *no_of_reps* value would be taken from the RACCU register.
- *rpt_step_value* - The step increment/decrement va

5.4 REFI_3 Instruction

REFI_3(rpt_delay_sd,rpt_delay, mode, outp_cntrl, fft_stage, refi_middle_delay_ext, no_of_rpt_ext ,rpt_step_value_ext ,fft_end_stage,DiMArch_Mode)

- rpt_delay - Number of clock cycles to wait before repeating the address pattern
- rpt_delay_sd - The rpt_delay is valid only if rpt_delay_sd is $'0'$ otherwise the rpt_delay would be taken from the RACCU register.
- $mode$ - The mode configures the valid DiMArch Rd/Write
- $outp\_cntrl$ - Selects the output port
- fft_stage - Configures the fft start stage (between 1 and 5).
- $refi\_middle\_delay\_ext$ - Configures the extended middle delay.
- $no\_of\_rpt\_ext$ - Programs the number of extended repeats.
- $rpt\_step\_value\_ext$ - Configures the extended repeat step value
- fft_end_stage - Configures the fft end stage (between 1 and 5, but greater than fft_stage).
- $DiMArch\_Mode$ - Selects the data transfer mode (between RFiles or RFile and Memory)

5.5 DELAY Instruction

DELAY (del_cycles_sd, del_cycles)

- del_cycles - Number of clock cycles to wait before decoding the next instruction
- del_cycles_sd - The del_cycles is valid only if del_cycles_sd is set.

5.6 RACCU Instruction

RACCU (raccu_mode , raccu_op1_sd, raccu_op1, raccu_op2_sd ,raccu_op2 , raccu_result_addrs)

- $raccu\_modes$ - Selects one of the RACCU modes.
- $raccu\_op1$ - Configures the raccu input #1 (operand).
- $raccu\_op1\_sd$ - The $raccu\_op1$ is considered if $raccu\_op1\_sd = 0$ otherwise the operand will be taken from the data register.
- $raccu\_op2$ - Configures the raccu input #2 (operand).

- $raccu\_op2\_sd$ - The $raccu\_op2$ is considered if $raccu\_op2\_sd = 0$ otherwise the operand will be taken from the RACCU data register.
- $raccu\_result\_addrs$ - Contains the loop register address.

5.7 LOOP_HEADER Instruction

LOOP_HEADER (index_raccu_addr , index_start, iter_no_sd, iter_no)

- $index\_raccu\_addr$ - Configures the RACCU address
- $index\_start$ - Configures the starting address
- $iter\_no\_sd$ - When '0', RACCU considers the input – fix it
- $iter\_no$ - Configures the iteration number

5.8 LOOP_TAIL Instruction

LOOP_TAIL (index_step, pc_togo, index_raccu_addr)

- $index\_step$ - Configures the loop increment value
- pc_togo - Configures the beginning of the loop
- $index\_raccu\_addr$ - Configures the RACCU address

5.9 SWB Instruction

SWB (from_block, from_address, from_port, to_block, to_address, to_port)

- $from\_block$ - Configures the FROM block
- $from\_address$ - Configures the FROM address
- $from\_port$ - Configures the FROM RFile port
- to_block - Configures the TO block
- $to\_address$ - Configures the TO address
- to_port - Configures the TO RFile port

5.10 BRANCH Instruction

BRANCH (brnch_mode, brnch_false_addr)

- $brnch\_mode$ - The conditional branch jumps to the false address when the ($brnch\_mode$ && seq_cond_status) == "00"
- $brnch\_false\_addr$ - Configures the false address

5.11 JUMP Instruction

JUMP (true_addrs)

- $true\_addrs$ - The PC will be configured with this value

5.12 SRAM_READ Instruction

SRAMAGU_read(mode, Initial_Address, Initial_Delay, Loop1_Iterations, Loop1_Increment, Loop1_Delay, Loop2_Iterations, Loop2_Increment, Loop2_Delay, sram_Initial_address_sd, sram_Loop1_iteration_sd, sram_Loop2_iteration_sd, sram_initial_delay_sd, sram_Loop1_delay_sd, sram_Loop2_delay_sd, sram_Loop1_increment_s
sram_Loop2_increment_sd)

5.13 SRAM_WRITE Instruction

SRAMAGU_write(mode, Initial_Address, Initial_Delay, Loop1_Iterations, Loop1_Increment, Loop1_Delay, Loop2_Iterations, Loop2_Increment, Loop2_Delay, sram_Initial_address_sd, sram_Loop1_iteration_sd, sram_Loop2_iteration_sd,sram_initial_delay_sd, sram_Loop1_delay_sd, sram_Loop2_delay_sd, sram_Loop1_increment_sd, sram_Loop2_increment_sd

- $mode$ - Configures the read, write operations
- $Initial\_Address$ - Configures the initial address
- $Loop1\_Iterations$ - Holds the number of iteration for the loop1
- $Loop1\_Increment$ - Configures the Loop1 increment value
- $Loop1\_Delay$ - Configures the Loop1 delay
- $Loop2\_Iterations$ - Holds the number of iteration for the loop2
- $Loop2\_Increment$ - Configures the Loop2 increment value

- $Loop2\_Delay$ - Configures the Loop2 delay
- $sram\_*\_sd$ - Corresponding values will be considered if it is $'0'$ otherwise it would be taken from the RACCU register.

5.14 ROUTE Instruction

ROUTE (Source_Row, Source_Col, Destination_Row, Destination_Col, DRRA_SEL, Union_Flag, Union_Port, origIn)

- $Source\_Row$ - Configures the source Row address
- $Source\_Col$ - Configures the source Column address
- $Destination\_Row$ - Configures the Destination Row address
- $Destination\_Col$ - Configures the Destination Column address
- $DRRA\_SEL$ - Configures the DRRA Row, when '0' Routing happens between the DRRA Row #0 and SRAM otherwise DRRA Row #1 and SRAM
- $Union\_Flag$ - Configures the DiMArch in Unicast mode when Set
- $Union\_Port$ - Configures the union port
- $origin$ - When '0' Destination initiates the routing else the source initiates the routing.

5.15 HALT Instruction

HALT

6 Runtime Address Constraint Computational Unit - RACCU

The DRRA has a dedicated hardware to calculate synchronization delays, address limits/bounds (AB) and address functions at run time. RACCU is controlled by the local DRRA sequencer and the output of the RACCU modifies the instructions which can be interpreted by the sequencer. Since the RACCU communicates only to the sequencers, this auxiliary hardware is placed in the sequencer in the design hierarchy. The RACCU consists of a storage block and a sequencer computational unit (SCU). The SCU has a dedicated multiplier, adder and a shifter to compute the synchronization delays, address limits and address functions dynamically.

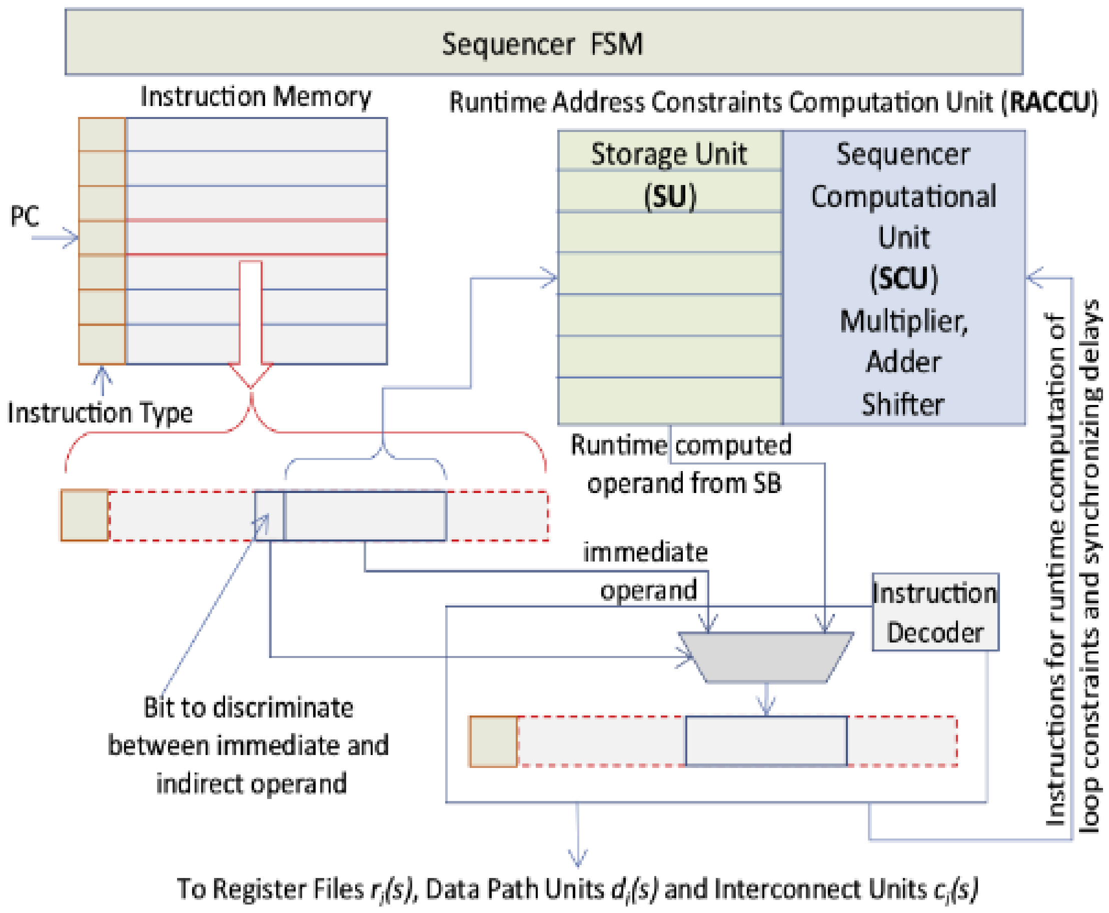

Sequencer FSM
Instruction Memory
Runtime Address Constraints Computation Unit (RACCU)
PC
Storage Unit (SU)
Sequencer Computational Unit (SCU) Multiplier, Adder Shifter
Instruction Type
Runtime computed operand from SB
immediate operand
Instruction Decoder
Bit to discriminate between immediate and indirect operand
Instructions for runtime computation of loop constraints and synchronizing delays
To Register Files $r_i(s)$, Data Path Units $d_i(s)$ and Interconnect Units $c_i(s)$

Figure 21: The RACCU Block

The figure 21 shows the working of the RACCU for an arbitrary DRRA sequencer instruction with a single indirect, i.e., runtime computed parameter. When a parameter is tagged as indirect by its direct/indirect discriminating bit, the content of the indirect parameter is an address in the SB where the runtime computed value is stored. This implies that this runtime computation must happen before fetching it. This is neatly handled by computing these parameters for the next iteration i + 1 in parallel, while the Functional Computation for iteration i is taking place. The primary reason for architecting an additional computational unit in form of SCU is to be able to do these two computations in parallel. The SB has registers to hold the loop indices/state, address function values and delay values. The number of registers is a design time decision that depends upon the max number of Address Functions and independent variables that the Address Functions could depend upon and the address space of the DRRA register files and DiMArch SRAM banks.

AGUs can only compute address functions; the address limits are configured by the sequencer instructions and they are parametric constants or static values. When the ABs are dynamically changing for each loop iteration, address bounds are computed by the RACCU and the AGUs are programmed with the computed address bounds. Unlike ABs, the AGU can calculate address functions independently until they are affine functions of a maximum of two variables corresponding to the indices of the two immediate enclosing loops. When an address function is non-affine and/or a function of more than two variables, the RACCU is used as a complementary resource.

RACCU has a few limitations which is discussed below. The register file depth of the RACCU is 8 which limits the number of independent variables and the number of nested loops. Another limitation of the RACCU is the arithmetic operations supported by it. Currently, the RACCU does not support the division operation. If the address computation (AC) requires expensive arithmetic computations like division, it has to be computed sequentially with the available arithmetic and logical operators.

6.1 List of Signals

Table 7: Switchbox Signals

Signal Name	Width	Description
clk	1	Synchronous clock to the RACCU
rst	1	Asynchronous reset
$raccu\_in1\_sd$	1	When set to '0' raccu_in1 will be considered, else the input will be taken from the data_register
$raccu\_in1$	6	RACCU input#1
$raccu\_in2\_sd$	1	When set to '0' raccu_in1 will be considered, else the input will be taken from the data_register
$raccu\_in2$	6	RACCU input#2
$raccu\_cfg\_mode$	4	Configures different RACCU modes
$raccu\_res\_address$	3	Contains the loop register address.
$raccu\_regout$	[3][6]	RACCU Data Register
$raccu\_loop\_reg$	3	RACCU Loop Register

6.2 RACCU Modes

Table 8: RACCU Modes

Signal Name	**Mode**	**Description**
$RAC\_MODE\_LOOP\_HEADER$	1	RACCU Loop Header mode
$RAC\_MODE\_LOOP\_TAIL$	2	RACCU Loop Tail mode
RAC_MODE_ADD	3	RACCU performs addition operation
RAC_MODE_SUB	4	RACCU performs SUB operation
$RAC\_MODE\_SHFT\_R$	5	Logical Shift Right
$RAC\_MODE\_SHFT\_L$	6	Logical Shift Left
$RAC\_MODE\_ADD\_WITH\_LOOP\_INDEX$	7	ADD + Loop Index

7 Switchbox

Switchbox The DRRA data interconnect has output buses driven by the DPUs and register files. These buses, placed horizontally between the rows are segmented wires that span N (=2) columns on each side. These horizontal output buses are intercepted by the vertical buses, which are the inputs to the DRRA cells. A configurable switch box is placed at their intersection as shown in the figure 22. Because of the sliding window connectivity, the switch box can get a maximum of 14 output lanes, 7 from each side. A lane represents a specific input/output of a register file or a DPU. A switchbox is composed of twelve 14x1 tri-stated multiplexers as shown in Figure X. Every column has four such switchboxes connected to 12 lanes to provide inputs to two DPUs with four inputs each and two register files with two inputs each. The size of each switchbox is 14x12 and they are mutually exclusive i.e. only one switch can write on a bus at a time. The configuration memory decides which output lanes connect to which input lanes. When a specific input lane is not driven by any output lane, it is tri-stated by the switchbox.

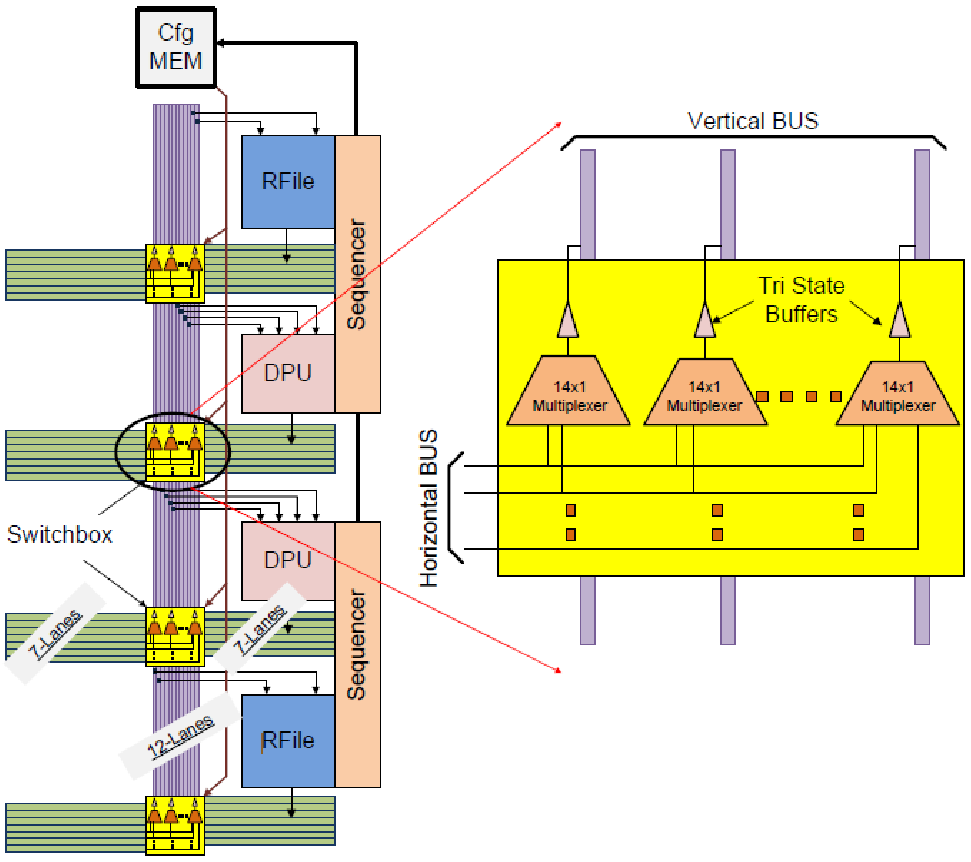


Figure 22: DRRA Interconnection Network, a) DRRA Interconnection Network with the horizontal and vertical buses. The signal from horizontal bus is connected to the vertical bus through a switchbox. b)Zoomed-in view of the switchbox

Configuration Memory - The configuration for all the switchboxes in a columns are stored in a small 12x6 bit configuration memory local to the lane and accessed by the two local sequencers. The 12 rows correspond to the 12 vertical buses and the 6 bits in a row is the configuration for one vertical bus corresponding to that row. In the beginning of the execution of a program, the sequencer configures the switchboxes to connect the inputs and outputs of its local RFile/DPU to other RFile/DPU. It may also reconfigure the switchboxes during the execution of the program to implement a ping-pong buffer or to satisfy a conditional execution which may require a change in input or output connectivity. The switchbox is partially re-configurable in one cycle. A DRRA column has four switchboxes which can write on a single bus. To protect from scenarios where more than one switchbox may write on a single bus at the same time, we have added two extra bits in every row of the configuration memory. These bits decides which switchbox will write on the bus and the outputs of others will stay tri-stated for that bus. This way the programmer does not have to worry about more than two switchboxes writing on the same bus.

7.1 List of Signals

Table 9: Switchbox Signals

Signal Name	**Width**	**Description**
inputs_reg	h_bus_ty(0 TO M-1, 0 TO 15)	Input bus from the RFile
inputs_dpu	h_bus_ty(0 TO M-1, 0 TO 15)	Input bus from the DPU
sel_r_int	s_bus_switchbox_ty	selects internal signals
sel_r_ext_in	s_bus_switchbox_ty	selects external signals
int_v_input_bus	v_bus_ty	internal out bus
ext_v_input_bus_out	v_bus_ty	external out bus

8 DiMArch

Distributed Memory Architecture (DiMArch) can provide high-bandwidth, Low-latency, scalable and reliable communication which is designed for DRRA. The following five features offered by the DiMArch.

- **Distributed:** DRRA being a fabric, the conpuatution is distributed across the chip which runs several applications in parallel, which distributed memory, the proposed deisgn enables multiple private and parallel excution environment.
- **Partitioning:** Due to the feature of distributed, the DiMarch enables the compile time repartitioning.
- **Streaming:** Each partitions includes the memory banks (mBanks) can be considerd as node and stream data to computation units. That enable elasticity in streaming by modify the delay valuse.
- **Energy Optimazation and Performance:** The DiMArch provide a optimazied path between memory and computation. That lower the latency and save the power supply by manage the unused nodes.
- **Scalability:** DiMArch is scalable with clock frequency and size of the network.

The current DiMArch fabric is composed of memory banks (mBanks), a circuit-switched data Network-on-chip (dNoC) as a connection of the mBanks and RFile of DRRA, a packet switched instruction Network-on-chip (iNoC) to create partitions streaming data and transport instructions from sequencer of the DRRA to instruction Switch. The fabric structure is shown in Figure 1.

8.0.1 Instruction Switch

The Instruction Switch(iSwitch) plays three distinct roles in the DiMArch architecture namely, it configures the partitions, represents the segmented buses and routes the instructions from source to the destination using the segmented buses. The iSwitch also connects to the other iSwitch in all four directions NORTH, EAST, SOUTH and WEST. The Instruction Switch has the following major components for each direction.

1 Instruction Decoder

2 Source Finite State Machine (FSM)

8.0.2 Instruction Decoder

An instruction packet contains the source, intermediate, and destination addresses along with an intermediate flag (Rt-flag) and an origin node flag. The Instruction decoder decodes the instruction received from a horizontal/vertical segmented bus and creates a path between the source and the destination. An origin node issues the instruction packet, which can be in the source or destination column. Initially, the source decoder checks if the instruction is direct or indirect using Rt-flag0 of the 3-bit Rt-flag. A direct instruction compares the source, destination and current node addresses while an indirect instruction compares the segment flag (Rt-flag1) and origin flag (Rt-flag2) along with the source, destination and intermediate node addresses. The segment flag helps to determine whether the packet is between the origin and the intermediate nodes or vice versa. In the latter condition the segment flag is flipped at the intermediate node. Currently, the source decoder supports only the direct instruction mode. The source decoder decides the source direction and also provides the source, destination and retransmission flags to the source FSM. For corner nodes, the retransmission flag is set. The source decoder also determines the partition directions which will be used by the source FSM.

As mentioned earlier, the iSwitch has four source decoders one for North, East, West and South directions. The iSwitch is designed such that, always the south/bottom decoder receives the instruction and begins decoding the instruction. Based on the source, destination and intermediate addresses, the source decoder connects the source and the destination nodes. The origin node flag decides whether the source or destination node needs to initiate the routing. As a design decision, for $SRAM\ Read$ the source node initiates the routing and for $SRAM\ Write$ the destination node initiates the routing.

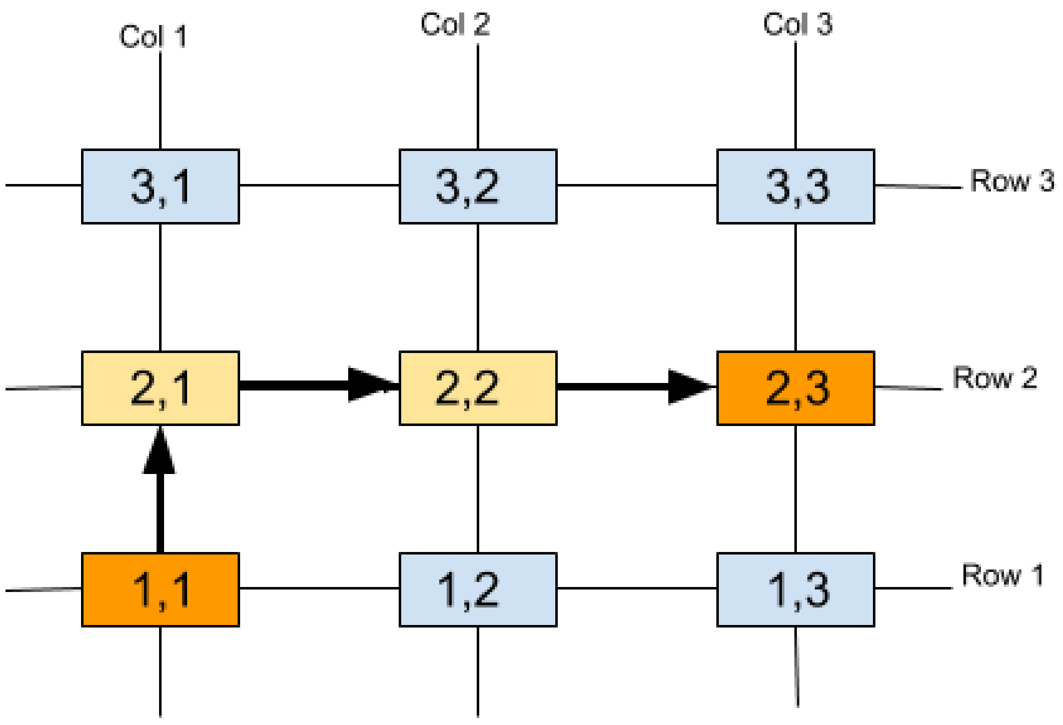


Figure 23: Routing from the source Node to the Destination Node

Consider an example shown in the figure 23 where the source node (1,1) has to transfer the data to the destination node (2,3). Based on the origin node flag, the source, destination and intermediate addresses in the instruction, the source decoder generates a command/instruction to its own iSwitch, so that the current iSwitch connects to one of the neighboring nodes. Interconnection rules for the instruction decoder will be discussed in the following sections.

The routing happens when the source IDs and destination IDs are different. As a result, there are eight possible cases listed in the table 10 below.
S_C - current column number of source node
D_C - current column number of destination node
S_R - current row number of source node
D_R - current row number of destination node
M_C - current column number of intermediate node
M_R - current row number of intermediate node

Table 10: Possible Source and Destination locations

Case	**Description**
1	$S\_R < D\_R$ and $S\_C < D\_C$
2	$S\_R < D\_R$ and $S\_C = D\_C$
3	$S\_R < D\_R$ and $S\_C > D\_C$
4	$S\_R = D\_R$ and $S\_C < D\_C$
5	$S\_R = D\_R$ and $S\_C > D\_C$
6	$S\_R > D\_R$ and $S\_C < D\_C$
7	$S\_R > D\_R$ and $S\_C = D\_C$
8	$S\_R > D\_R$ and $S\_C > D\_C$

An intermediate node which receives the data vertically, sets the corner_node flag when the data has to be sent to the left or right, i.e., for sending data from the NORTH to the EAST or WEST. We refer to such intermediate iSwitches as $turnaround$.

The path creating rule is explained in detail below: Consider the case 1 $S\_R < D\_R$ and $S\_C < D\_C$ for the path creating rule:

If the current iSwitch is a Source node, the iSwitch reads data from the memory and writes it to the NORTH node. It also sets the source_flag.

If the current iSwitch is a destination node, the iSwitch reads data from the west node and writes it to the memory. It also sets the destination_flag.

If the current iSwitch belong to one of the three situations below, it will be considered as an intermediate node (I):

1 When $M\_R = D\_R \& S\_C < M\_C < D\_C$, the iSwitch sets the direction from the WEST node to the EAST node, i.e, the iSwitch reads from the WEST node and writes to the EAST direction.

2 When $M\_C = S\_C \& S\_R < M\_R$, the iSwitch sets the direction as SOUTH To NORTH

3 When $M\_C = S\_C \& M\_R = D\_R$, the iSwitch sets the direction as SOUTH To EAST

In all other cases the iSwitch will be insensitive to the current instruction.

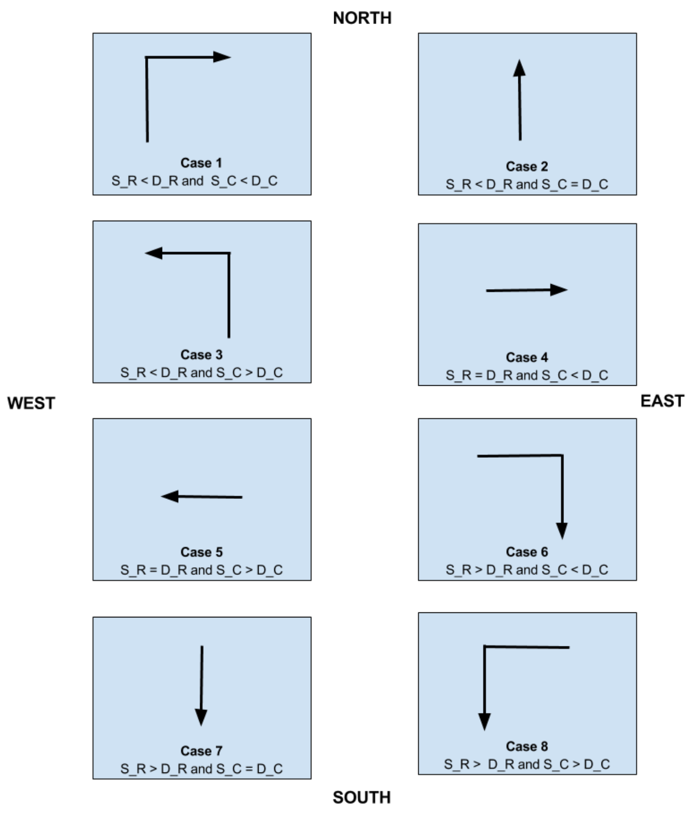


Figure 24: source decoder routing rule

8.0.3 Source FSM

The Source decoder sets the direction of the data, the source flag, destination flag and the corner flag. The source FSM analyses the source decoder flags and either routes, partitions or performs both functionalities simultaneously. In the routing state, the FSM checks if it is a source, destination, intermediate node or corner node. If the source flag or destination flag is set, the FSM moves to the WAIT4AGU state and waits for the AGU instruction. The received AGU instruction is then sent to read AGU or write AGU. In the partitioning state, according to the direction received from the source decoder, the FSM makes sure the path is open, otherwise the FSM opens the path. Again, if the source or destination flag is set, the FSM does the needful. If the retransmission flag is true, then the received instruction will be retransmitted either to the vertical or horizontal segmented bus. If the instruction is received from the vertical or horizontal bus segment and the retransmit flag is set, then the instruction will be sent to the vertical or horizontal bus segments, respectively. Retransmission happens after the partitions are setup. Figure 25 shows source FSM of DiMArch.

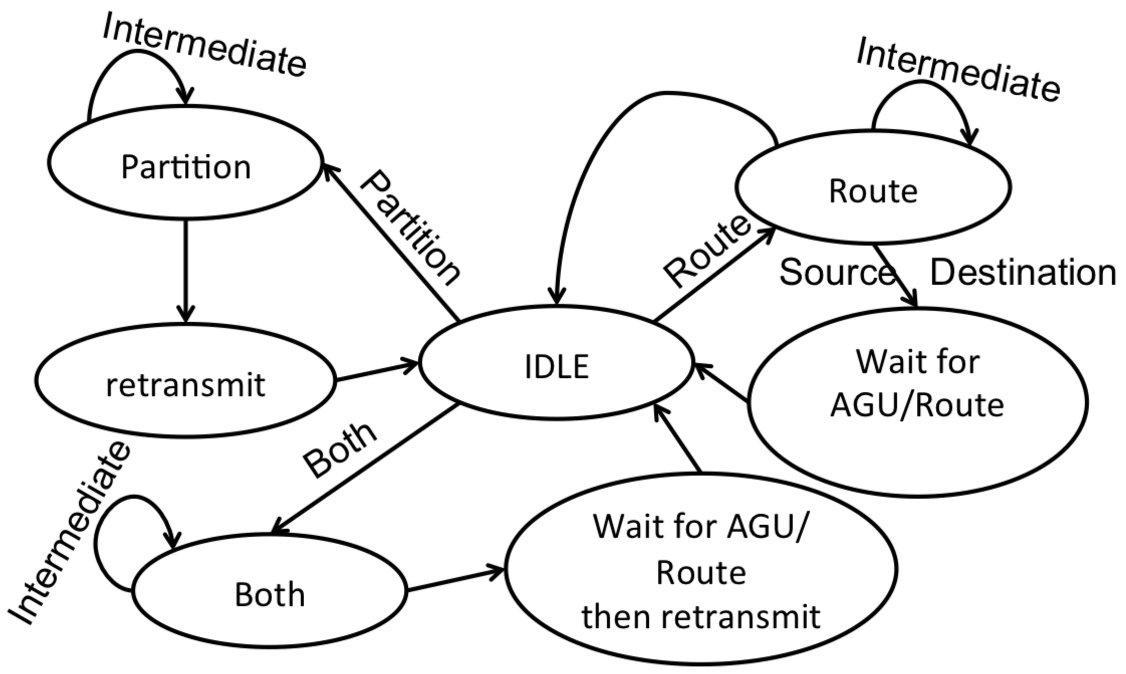


Figure 25: source FSM

8.0.4 SRAM Banks and AGU

The memory bank is SRAM macros, typically 2 KB, a design time decision, as the goal is to align memory bank with the memory manager. Memory is controlled by the SRAM AGU. The memory bank receives the address select, r/w and enable signals. The sequencer provides a general purpose timing model using three delays, An initial delay before a loop, an intermittent delay before every r/w within a loop and an end delay at the end of loop before repeating the next iterations. These delays are used to synchronize the memory to register file streams with the computation as discussed in the DRRA AGU sections. Individual delays can be changed computation which makes streaming elastic.

The SRAM has two AGUs one for the SRAM read and another one for the SRAM write operation. The working principle of the SRAM AGU is similar to the DRRA AGU.