dRMT Compiler

Table of Contents

[General Purpose ALU 2](#_Toc505077876)

[Overview: 2](#_Toc505077877)

[Target Implementation: 2](#_Toc505077878)

[Compiler Implementation: 3](#_Toc505077879)

[Keygen Module: 5](#_Toc505077880)

[Overview: 5](#_Toc505077881)

[Target Implementation: 5](#_Toc505077882)

[Compiler Implementation: 6](#_Toc505077883)

# General Purpose ALU

## Overview:

dRMT is special case Register Memory Architecture where operations are performed on the Memory directly. There are no special registers. The ALU module is the primary computational unit in the dRMT platform. Each processor has up to 32[**NUMALU**] general purpose ALUs. All these ALUs working in parallel allows dRMT to exploit the inherent parallelism that might exist in the program.

## Target Implementation:

ALU

Byte Vector

Bit Vector

PHV

Input Selector

Scratch Pad

Config (Constants)

The above figure shows how each general purpose ALUs work. Each ALU can have up to 2 operands. The instruction is of the following form

**[op0\_type(2bit), op0\_len(3bit), op0\_off(8bit), op1\_type(2bit), op1\_len(3bit), op1\_off(8bit), op\_code(5bit), res\_len(3bit), res\_off(8bit), pred\_en(1bit), pred\_off(8bit)]**

Each Operand can be selected from 3 different sources each addressable separately. Seen as opx\_type in the instruction format.

1. **Packet Header Vector [PHV]** - This is where the Packet data is present.
2. **Scratch Pad** – This is where the Match Data is staged.
3. **Config** – This is where the programmable constants (from the input P4 program) are stored.

PHV and Scratch pad have 2 separate sections.

1. **Byte Addressable section** – As the name suggests, this allows for byte granular access to Data.
2. **Bit Addressable section** – This portion allows bit granular access.

The result of the Byte ALUs are always stored in the PHV. In other words, the destination is always PHV. The res\_off and opx\_off are locations of the result and operands in their respective address spaces.

Each Operand is further categorized based on its length, seen as opx\_len and res\_len in the instruction format.

1. **Bit Length** – This type of operand is of size 1 Bit and can only exist in the Bit Addressable section of the Operand Source.
2. **1 Byte Length** – This is of size 1 Byte and can exist in the Byte Addressable section.
3. **2 Byte Length** – Same as above and of size 2 Byte [16 bit].
4. **4 Byte Length** – Same as above and size of 4 Byte [32 bit].

The following are the op\_codes that are supported.

1. **NOOP** – No operation
2. **COPY** – Has only 1 operand. Res = Op0
3. **ADD** – Res = Op0 + Op1. [No carry and underflow bits support]
4. **SUB** – Res = Op0 – Op1. [No carry and underflow bits support]
5. **XOR** – Res = Op0 ^ Op1. [Bitwise XOR]
6. **AND** – Res = Op0 & Op1. [Bitwise AND]
7. **OR** – Res = Op0 | Op1. [Bitwise OR]
8. **SHL** – Res = Op0 << Op1. [Shift Left. Only 5 LSB of Op1 is considered]
9. **EQ** – Res = Op0 == Op1. [Equality test. Result should be set to Bit vector of PHV]
10. **GT** – Res = Op0 > Op1. [Greater than. Result should be set to Bit Vector of PHV]
11. **LT** – Res = Op0 < Op1. [Less than. Result should be set to Bit Vector of PHV]
12. **NE** – Res = Op0 != Op1. [Inequality test. Result should be set to Bit Vector of PHV]

Each instruction in dRMT is predicated. The pred\_en is a single bit which indicates if the instruction is predicated or not. The pred\_off indicates where the predicate value is taken from the **PHV** [Bit vector portion].

## Compiler Implementation:

The major job of the compiler as far as programming the ALUs are concerned is to transform P4 expressions/assignment statements to the dRMT ALU instruction. The following are mainly the operations that the compiler performs to achieve that

1. Transforming to Quadruples – In this step, complex expressions are broken down to simple expressions. This is achieved by introducing temporary variables and introducing intermediate statements.
2. Predicate Calculation – In this step, each statement is associated with a predicate bit.

# Keygen Module:

## Overview:

The main responsibility of this module is to create the keys required to be sent on the Key Bus to the Sea of Memory(SOM). The module needs to be programmed to be able to copy the Key data from the Packet Vector Buffer (PHV) to the Key Bus.

## Target Implementation:

This section briefly describes the module implementation.

Byte Vector

Bit Vector

PHV

Config

Keygen

SOM#0

SOM#1

SOM#2

XBAR

dRMT Processor#0

Fig. 1. This shows that the key being formed from the PHV and sent across 2 segments to the XBAR which in turn sends it to SOM#0.

The data in DRMT resides in the PHV so the Keygen configuration has to

## Compiler Implementation:

In this section, let us take an example of a P4 table. Lets refer to the source code below.

@name("table2") table table2 { // HW table Id of 0x2  
 actions = {  
 action2;  
 @default\_only NoAction;  
 }  
 key = {  
 hdr.ethernet.dstAddr: exact;

hdr.ethernet.srcAddr: exact;  
 }

size = 2048;  
 default\_action = NoAction();  
}

As we can see the table is an exact map table with Ethernet Source and Destination Address as the key. The source and destination address are 6 Bytes each which means it is a 12 Byte long key. This key is supposed to be sent over the key bus.

The way dRMT allows this to enable a config register called ybyt [Source Byte]. Let us assume the key bus is 10 bytes [**NUMKBYT**] long and the number of segments is 4[**NUMKSEG**]. Then to send a 12 byte key we would need to use at least 2 segments, using any more is not needed and would be wasteful.

The ybyt config register expects an address from where the byte needs to be copied from in the PHV [Byte vector]. This is copied to form the key. Let us assume that the Ethernet Source Address is present at location 0x0 and Destination address is present at 0x8. The ybyt register will be programmed as follows.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0x0 | 0x1 | 0x2 | 0x3 | 0x4 | 0x5 | 0x8 | 0x9 | 0xa | 0xb |
| 0xc | 0xd | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 |
| 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 |
| 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 |

Config Register ybyt is programmed as shown above. As we can see this particular match uses 2 segments out of 4. The rest of the 2 segments are not being used in this example match stage. Given this example we could have done matches for 2 more table assuming each of them could fit their key in 1 segment. A particular segment can never be shared across multiple tables.

To associate a table Id to a segment and to indicate if the segment contains a valid key or not, there are 2 more config registers ktbl and kvld.

Their programming is shown below.

|  |
| --- |
| 0x2 |
| 0x2 |
| 0x0 |
| 0x0 |

Config register of ktbl associates segment 0 and 1 to table 2. A thing to remember is that tables which uses more than 1 segment, need not use 2 consecutive segments. Here it would have been perfectly legal to use segments 1 and 3 as well.

|  |
| --- |
| 0x1 |
| 0x1 |
| 0x0 |
| 0x0 |

Config register of kvld says that only segment 0 and 1 are being used. One note here is that the vld bit is set at segment level only and not and ybyt level. So key generated is always a multiple of **NUMKBYT**. The assumption is that the table will be programmed appropriately.