

Project Zipline Prefix

Micro Architecture Specification

**Authors:**

**Microsoft Corporation**

**Broadcom Corporation**

**Revision History**

|  |  |
| --- | --- |
| Date | Description |
| 04/03/2019 | Version 1.0 |
|  |  |

**License**

Contributions to this Specification are made under the terms and conditions set forth in the Open Web Foundation Contributor License Agreement (“OWF CLA 1.0”) (“Contribution License”) by:

Microsoft Corporation

Broadcom Corporation

Usage of this Specification is governed by the terms and conditions set forth in Open Web Foundation Final Specification Agreement (“OWFa 1.0”) (“Specification License”).

**Note**: The following clarifications, which distinguish technology licensed in the Contribution License and/or Specification License from those technologies merely referenced (but not licensed), were accepted by the Incubation Committee of the OCP:

**None.**

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY LIABILITY FOR LOST PROFITS OR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT, SPECIAL OR PUNITIVE DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND EVEN IF OCP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

[Overview 7](#_Toc5181540)

[1.1 Prefix Engine Block Diagram 7](#_Toc5181541)

[1.2 Interfaces 8](#_Toc5181542)

[1.3 Data Flow Description 10](#_Toc5181543)

[1.3.1 Inbound Data Flow 11](#_Toc5181544)

[1.3.2 Feature Extraction 11](#_Toc5181545)

[1.3.3 Prefix Selection 12](#_Toc5181546)

[1.3.4 Prefix Engine Outbound Data Flow 12](#_Toc5181547)

[1.3.5 Control Data Flow 12](#_Toc5181548)

[1.3.6 Prefix Frame Header Modifications 13](#_Toc5181549)

[1.4 Performance 13](#_Toc5181550)

[1.4.1 External Prefix Engine Bandwidth Requirements 13](#_Toc5181551)

[1.4.2 Internal Prefix Engine Bandwidth Requirements 13](#_Toc5181552)

[1.4.3 Latency 14](#_Toc5181553)

[1.4.4 Throughput 14](#_Toc5181554)

[2 Detailed Description 15](#_Toc5181555)

[2.1 Feature Extractor 15](#_Toc5181556)

[2.1.1 Feature Counter 16](#_Toc5181557)

[2.1.2 Feature Extractor Block Diagram 17](#_Toc5181558)

[2.2 Programming Feature Extractor Configuration 19](#_Toc5181559)

[2.3 Recognizer uSequencer 24](#_Toc5181560)

[2.3.1 Instruction Set 24](#_Toc5181561)

[2.3.2 List of Instructions 25](#_Toc5181562)

[2.3.3 Pipeline Block Diagram 28](#_Toc5181563)

[2.3.4 Program Counter 29](#_Toc5181564)

[2.3.5 Instruction Memory 29](#_Toc5181565)

[2.3.6 Input Registers 29](#_Toc5181566)

[2.3.7 Local Registers 30](#_Toc5181567)

[2.3.8 Source Address Table Memory 30](#_Toc5181568)

[2.3.9 Crossbar unit (xbar) 30](#_Toc5181569)

[2.3.10 Coefficient Table Memory 31](#_Toc5181570)

[2.3.11 ALU 31](#_Toc5181571)

[2.3.12 Activation Function 32](#_Toc5181572)

[2.3.13 Neuron Sort 35](#_Toc5181573)

[2.3.14 Prefix Store 36](#_Toc5181574)

[2.3.15 Pipeline stalls 36](#_Toc5181575)

[2.3.16 Errors 36](#_Toc5181576)

[3 Debug 37](#_Toc5181577)

[3.1 Statistics 37](#_Toc5181578)

[3.2 Control Status Registers (CSR) 37](#_Toc5181579)

[3.3 uSequencer Debug Mode 37](#_Toc5181580)

[4 Appendix 39](#_Toc5181581)

[4.1 Feature Extractor Psuedo code 39](#_Toc5181582)

[4.2 Sigmoid Activation Function 46](#_Toc5181583)

[4.3 Program Example 47](#_Toc5181584)

Table of Figures

[Figure 1 Prefix Engine Block Diagram 7](#_Toc5181585)

[Figure 2 Top Level Block Diagram 11](#_Toc5181586)

[Figure 3 Feature Counter 17](#_Toc5181587)

[Figure 4 Feature Extractor Block Diagram 18](#_Toc5181588)

[Figure 5 Pipeline Block Diagram 28](#_Toc5181589)

[Figure 6 One input per neuron 49](#_Toc5181590)

[Figure 7 Two inputs per neuron 50](#_Toc5181591)

[Figure 8 Four inputs per neuron 51](#_Toc5181592)

[Figure 9 Three inputs per neuron, second level 53](#_Toc5181593)

Table of Tables

# Overview

The Prefix Engine, when enabled for XP10 Prefix Mode, examines the incoming data to determine the pre-existing prefix data that should be used to seed the compression engine to achieve the best compression.

A standard compression prefix is selected by examining and classifying the first 4kB (or less) of data prior to starting compression.  The standard prefixes and the classifier parameters will be loaded into the chip at initialization time.  The Prefix engine supports up to 63 standard prefixes of 1kB size, prefix number 0 is reserved for “no prefix”.

It is also possible for the user to provide the prefix data and is referred to as “user prefix”, between 1KB and 64KB in size. User prefix is provided from the host memory as literal data by the user as part of a compression or decompression command.  When User Prefix Data is used, this engine does not modify the user prefix or add any additional prefix data.

The decompression prefix is the same as the compression prefix and will be communicated via header or may be selected by the user (whether standard or user prefix).

The Prefix Attach is responsible for inserting the selected prefix data into the data stream. Note, this document covers the Prefix Engine only; the Prefix Attach is covered by a separate document.

## Prefix Engine Block Diagram



Figure Prefix Engine Block Diagram

## Interfaces

The Prefix Engine external interfaces are shown below.

| Name | I/O | Description |
| --- | --- | --- |
| **Clocks and resets** |  |  |
| clk | I | 800MHz clock |
| rst\_n | I | Active low reset |
| **AXI4S Slave** |  |  |
| tvalid | I | TVALID indicates that the master is driving a valid transfer.  A transfer takes place when both TVALID and TREADY are asserted. |
| tlast | I | TLAST indicates the boundary of a frame. |
| tid | I | TID is the data stream identifier that indicates different streams of data. |
| tkeep[7:0] | I | TKEEP is the byte qualifier that indicates whether the content  of the associated byte of TDATA is processed as part of the data stream.  Associated bytes that have the TKEEP byte qualifier de-asserted are null bytes and can be removed from the data stream. |
| tuser[7:0] | I | TUSER is user defined sideband information that can be transmitted alongside the data stream. |
| tdata[63:0] | I | TDATA is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes. |
| tready | O | TREADY indicates that the slave can accept a transfer in the current cycle. |
| **AXI4S Master** |  |  |
| tvalid | O | TVALID indicates that the master is driving a valid transfer.  A transfer takes place when both TVALID and TREADY are asserted. |
| tlast | O | TLAST indicates the boundary of a frame. |
| tid | O | TID is the data stream identifier that indicates different streams of data. |
| tkeep[7:0] | O | TKEEP is the byte qualifier that indicates whether the content  of the associated byte of TDATA is processed as part of the data stream.  Associated bytes that have the TKEEP byte qualifier de-asserted are null bytes and can be removed from the data stream. |
| tuser[7:0] | O | TUSER is user defined sideband information that can be transmitted alongside the data stream. |
| tdata[63:0] | O | TDATA is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes. |
| tready | I | TREADY indicates that the slave can accept a transfer in the current cycle. |
| **RBUS In** |  |  |
| addr[31:0] | I | Rbus byte address |
| wr\_strb | I | Rbus write strobe indicates start of a write transaction |
| wr\_data[31:0] | I | Rbus write data |
| rd\_strb | I | Rbus read strobe indicates the start of a read transaction |
| rd\_data[31:0] | I | Rbus read data |
| ack | O | Rbus acknowledge indicates the end of a transaction |
| err\_ack | O | Rbus error acknowledge indicates an aborted transaction due to error |
| **RBUS Out** |  |  |
| addr[31:0] | O | Rbus byte address |
| wr\_strb | O | Rbus write strobe indicates start of a write transaction |
| wr\_data[31:0] | O | Rbus write data |
| rd\_strb | O | Rbus read strobe indicates the start of a read transaction |
| rd\_data[31:0] | O | Rbus read data |
| ack | I | Rbus acknowledge indicates the end of a transaction |
| err\_ack | I | Rbus error acknowledge indicates an aborted transaction due to error |

## Data Flow Description

Below is a top level block diagram for the CCEIP, with the Prefix Engine highlighted in blue, to show how it fits into the overall Project Zipline data flow:



Figure Top Level Block Diagram

### Inbound Data Flow

The inbound Data Frames are presented to the Prefix Engine via the Streaming AXI Slave interface. The data frames are written into a TLV Parser. The TLV Parser is a common block that can be instantiated in any Project Zipline Engine sub-design. The TLV Parser is used to split out the inbound TLV data stream into TLV’s of interest to the block’s main logic and those which the block doesn’t use are simply passed through. Additionally the TLV Parser will reassemble the two data streams back into one outbound AXI stream.

### Feature Extraction

The Feature Extractor will read the frame data from the TLVP inbound interface until the end of frame is reached. The feature extractor processes eight bytes of inbound frame data per clock cycle. The Feature Extractor examines the Frame Data and gathers feature counts to present to the Recognizer uSequencer. The Feature extract will always operate on the first 4KB of the frame’s data or up to the frame size in the event of smaller frames. This is not programmable. The block is broken up into 4x1kB sections of data. After the block size has been reached, the feature extractor stores the feature count in the designated input register of the Recognizer uSequencer. The Feature Extractor has a total of 256 unique features, organized as 64 features per 1kB data block. The Feature details are loaded into the Feature Extractor at initialization time. The Features must not be modified while the Feature Extractor is running or unpredictable results will be output.

Each feature is a count of some property in the data. For example,

* a count of how many times a character, say “x” is in the data
* a count of how many times a character string, say “xyz” is in the data (supports strings of length up to 4)

In addition, counts can also be obtained for statistics like:

* how many characters are less than or equal to character “x”?
* how many characters are greater than or equal to character “x”?

### Prefix Selection

The Recognizer uSequencer takes as input the counts accumulated in the feature extraction unit and using a neural network algorithm determines the index of the prefix block to be used by the match engine.

The Recognizer uSequencer is a programmable processor with a custom instruction set. The program is stored in an internal Instruction Memory. A typical 4-6 layer neural network can be implemented by this block. The exact number of layers that can be supported depends on the number of inputs in each layer. The output is a 6-bit integer which points to the selected prefix data to be used. If no prefix can be determined the prefix selector output will be set to zero.

### Prefix Engine Outbound Data Flow

The AXI Master I/F streams out the Frame data, when the Prefix Attach Engine indicates that it is ready to receive data. The Compression Header is updated with the prefix selector data as it is streamed out of the Prefix Engine.

### Control Data Flow

An Rbus Ring Interface will be provided to provide for Configuration & Status Register (CSR) access as well as various debug facilities. The registers provided by this interface will provide read/write access to to the following:

* AXI streaming interface statistics, i.e. data counters
* Feature Extractor individual feature match values, and chaining controls
* Recognizer uSequencer control/status and Instruction Memory
* Prefix Attach Data Store access, i.e. load prefix data
* Debug facilities

### Prefix Frame Header Modifications

Refer to the Project Zipline documentation for a description of Project Zipline TLV Headers.

#### CMD Header

The Prefix Engine will modify the inbound Command (CMD) Header to update the Prefix Selector field with the prefix selected by the Recognizer uSequencer.

#### Frame Footer TLV

If the Prefix Engine detects an error in the inbound Frame, the outbound Frame Footer TLV will be modified by updating the first.

## Performance

The Prefix Engine must be able to run at 25 Gbs for a 2KB frame and allocate a budget for the Feature Extractor, and Recognizer.

### External Prefix Engine Bandwidth Requirements

The bandwidth breakdown is as follows:

1. The 2KB frame needs to be processed in 512 clocks for 25Gbps performance.
2. Using a 50Gbps transport path a 2KB frame these 512 clock cycles are budgeted as follows (425 clocks):
3. 256 Clocks to transfer the 2KB frame through the Feature Extractor
4. 40 Clock to transfer the TLV stackup (estimated worst case budget)
5. 128 Clock to pre-pend the Prefix data (1024 Bytes)

### Internal Prefix Engine Bandwidth Requirements

Internally the Prefix engine is pipelined into 2 sections: Feature Extraction and Recognizer. Each of these blocks needs to be able to support accepting and consuming a new Frame every 512 clocks for line rate 2KB frame support.

The total budget internally for the Feature Extractor is as follows:

1. 256 Clocks for the feature Extractor (8 Bytes/clock with 2KB); Feature Counters need to be double buffered.

The total budget for the Recognizer is as follows:

1. 128 Clocks to pre-pend the Prefix Data (1024 Bytes) on 8 Byte outgoing datapath
2. 40 Clocks to transfer the RLV statckup on 8 Byte outgoing datapath
3. 16 Clocks of internal pipelining to move Recognizer registers
4. 328 Clocks remain available for the Recognizer program to run at line rate.

### Latency

The latency of the Prefix Engine is expected to be the following:

1. 256 Clocks for the Recognizer
2. 512 clocks for a maximal length Feature Extractor and Prefix Attach.

### Throughput

This block will support the line rate of at 25 Gbs for a 2KB frame.

# Detailed Description

## Feature Extractor

This block gathers feature counts over the selected block size. The block size is always 4kB of data unless the message is smaller. The block is broken up into 4x1kB sections of data for the feature accumulation.

There are total of 256 unique features possible, but only 64 features per 1kB data block. The features in a specific 1kB of data will be called a feature set, so there are up to four feature sets per frame. For smaller frames, features are only counted for valid data and entire feature sets are zeroed out, per the following rules:

  1kB data (or less)

* Set 0 : based on data
* Set 1: reset to 0
* Set 2: reset to 0
* Set 3: reset to 0

  1-2kB data

* Set 0: first 1kB of data
* Set 1:  1kB-2kB of data
* Set 2: reset to 0
* Set 3: reset to 0

2-3kB data

* Set 0: first 1kB of data
* Set 1:  second 1kB of data
* Set 2: 2kB-3kB of data
* Set 3: reset to 0

3kB or more

* Set 0: first       1kB of data
* Set 1:  second 1kB of data
* Set 2: third 1kB of data
* Set 3: fourth 1kB of data

Feature characteristics:

1. A feature works on a character.
2. A character is 8 bits (1 byte).
3. Each character is independent of all the other characters.
4. Each feature could be dependent on the previous feature/character (did it match or not)
5. A feature compare is one of four compare types (described in 2.2.1.1)
6. If the condition is true, the feature detection counter is updated.

### Feature Counter

A Feature Counter works on 8 characters in parallel per clock cycle. Each feature operates on a 1kB block of data at a time. When the end of the 1 kB is reached the feature counters are transferred to the recognizer input register.

Each feature is a three stage pipeline:

Stage 0: basic match

The input char is compared to one of four match values, one for each 1kB of data. The compare type (described in 2.2.1.1) is configurable. A match will load a logic 1 into the stage0 register, all other results will load a “0”.

Stage 1: prior check

If the use\_prior is configured then the match from stage0 is logically “anded” with the prior\_in. The output from each character (8 total) is summed, generating a number from 0 to 8.

Stage 2: count

The count from stage1 is added to the running count accumulator. The counter will saturate at 0xFF.



Figure Feature Counter

### Feature Extractor Block Diagram

The following Block Diagram shows the top level design of the Feature Extractor.

The Feature Extractor contains a state machine which is used to:

* Read input data from the AXI Interface FIFO and load into the Feature Group units
* Maintain a 1kB counter to count input characters and control the load of:
  + Feature counter configuration into each feature counter
  + Feature counts into the output registers at 1KB increments
* When the end of a data block is reached the feature counts are written into the feature counter FIFO’s. A separate FIFO is used for each 1KB counter for a total of 4 FIFO’s. The Feature Extractor will stall if any of these FIFO’s are full.

The Feature Extractor also instantiates 64 feature counters as described in the previous section.



Figure Feature Extractor Block Diagram

#### Feature Configuration

All feature characteristics are loaded into the Feature Extractor at reset/initialization time, while the Feature Extractor is disabled and must not be modified while the Feature Extractor is enabled. Each feature requires programming of a Feature Extractor Configuration Register, described in Section 2.2

Note: it is possible to program the configuration registers in combinations that would result in nonsensical or unusable counter results. The hardware does not prevent this condition.

## Programming Feature Extractor Configuration

The Feature Extractor hardware consists of 64 physical counters, where each counter can have a different configuration for each 1K bye of input data. This block implements 256 Features. At configuration time each feature must be configured. Each feature extractor configuration Register consists of the following:

#### Feature Extractor Configuration Register

| Name | Bit | Description |
| --- | --- | --- |
| use\_prior | 3 | This bit is always zero for features 0, 4, …, 252, as they are the start of a feature control register group.  0-      Ignore previous feature compare  1-      Use previous feature compare output. Previous feature match output must be true, and the match condition below must be true, for feature match output to be asserted for this feature. |
| no\_delay | 2 | 0- Delay match output by one character time. Outputs match value for previous character. Used for string matching.  1- Do not delay match output. Outputs match value for current character, as well as pass through of Prior Condition Matched. Used for range/set matching. |
| cmp\_type | 1:0 | 0- Equals the feature value (==)  1- Greater than or equal to the feature value (>=)  2- Less than the feature vale (<)  3- Equal to or pass through (pass |==). Match output is true if prior match input was true (and enabled), or if current character is equal to the feature value. (Used for character set matching). |
|  |  |  |

#### Feature Extractor Configuration Examples

In the following examples Comparator a,b,c,d refer to the 4 input comparators for each byte lane of the input character data

#### String matching

#### 2.2.1.3.1  4 char (e.g. DEAD)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b1000   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”E” | | 4’b1000   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b1000   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | |

#### 2.2.1.3.2 3 char (e.g. EAD)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”E” | | 4’b1000   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b1000   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | |

#### 2.2.1.3.3 2 char (e.g. AD)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”E” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b1000   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | |

#### Range matching: (e.g. A <= char <= Z)

#### 2.2.1.4.1  End range character excluded, e.g. A <= char < Z (N+2 value: start, N+3 value: end)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”X” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”X” | | 4’b0101   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 1 | | cmp\_type | 1(GTEQ) | | match\_val | ”A” | | 4’b1110   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 2 (LT) | | match\_val | ”Z” | |

#### End range character included, e.g. A <= char <=Z (N+2 and N+3 value set to end of range char)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b0101   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 1 | | cmp\_type | 1 (GTEQ) | | match\_val | ”A” | | 4’b1110   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 2(LT) | | match\_val | ”Z” | | 4’b1111   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 3(EQOR) | | match\_val | ”Z” | |

#### Character set matching

#### 2.2.1.5.1 Group of four (e.g. match on A or D or F or 5)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0100   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 1 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b1111   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 3 (EQOR) | | match\_val | ”D” | | 4’b1111   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 3(EQOR) | | match\_val | ”F” | | 4’b1111   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 3(EQOR) | | match\_val | ”5” | |

#### 2.2.1.5.2 Group of three (e.g. match on D or F or 5)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b0100   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 1 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b1111   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 3(EQOR) | | match\_val | ”F” | | 4’b1111   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 3(EQOR) | | match\_val | ”5” | |

#### 2.2.1.5.3 Group of two (e.g. match on F or 5)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b0100   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 1 | | cmp\_type | 0(EQ) | | match\_val | ”F” | | 4’b1111   |  |  | | --- | --- | | use\_prior | 1 | | no\_delay | 1 | | cmp\_type | 3(EQOR) | | match\_val | ”5” | |

#### Single character match (e.g. match 5)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0(EQ) | | match\_val | ”F” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0(EQ) | | match\_val | ”5” | |

#### Disable counter: (feature N+3 value set to zero)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0(EQ) | | match\_val | ”F” | | 4’b0010   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 2(LT) | | match\_val | ”0” | |

#### Count all characters: (feature N+3 value set to zero)

|  |  |  |  |
| --- | --- | --- | --- |
| Comparator a | Comparator b | Comparator c | Comparator d |
| 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”A” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0 (EQ) | | match\_val | ”D” | | 4’b0000   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 0(EQ) | | match\_val | ”F” | | 4’b0001   |  |  | | --- | --- | | use\_prior | 0 | | no\_delay | 0 | | cmp\_type | 1(GTEQ) | | match\_val | ”0” | |

## Recognizer uSequencer

This block takes as its inputs, the counts from the feature extraction unit. Using a neural network algorithm it determines the index of the prefix block to be used by the match engine.

### Instruction Set

The custom instruction format is defined as follows:

|  |  |  |
| --- | --- | --- |
| **Bits** | **Name** | **Description** |
| [23:22] | Reserved |  |
| [21:18] | Opcode | Currently 10 opcodes defined see below. Others unused. |
| [17:10] | Src1 | <Coefficient memory address | neuron count | shiftCount>  shiftCount is used in the activation functions |
| [9:2] | Src2 | <Source Address table pointer | prefixThreshold > |
| [1] | Src3 | <LR | IR>  1=Local Register , 0=Input Register |
| [0] | Src4 | <U | L>  1=Upper, 0=Lower |

### List of Instructions

The following table lists the instructions implemented in the uSequencer.

|  |  |  |
| --- | --- | --- |
| **Instruction:**  **Opcode, Operands** | **Opcode** | **Description** |
| Nop | 0000 | No operation.  Instruction needed to flush pipeline for layer changes |
| Halt | 0001 | This is the end of the program. We can now start a new operation. This instruction will FLUSH the pipeline by continuously inserting a NOP instruction, while waiting for the next program to start. This NOP will walk down the pipeline and set the pipeline registers to their initial state,  PC = 0 , Accumulator == 0, L0/L1 == 0, unlock |
| HOLD | 1010 | This instruction is similar to the HALT instruction except it holds the pipeline at its current state. The HOLD instruction will insert the NOP instruction, but because the pipeline is held while waiting for the next program the state of the pipeline registers is saved. Once the next program starts the NOP will proceed down the pipeline and reset the pipeline registers just like the HALT instruction. This instruction can be used interchangeably with the HALT instruction. This instruction can be useful for Design Verification and/or debugging purposes. |
| Unlock | 0010 | Once we start executing a new operation, the registers to that engine are locked . This instruction takes the place of the normal, NOP in the layer change. Once the unlock occurs the program should NOT read engine input as it may be changed. |
| ReLU\_store  Src1[7:0] - shiftCount  Src2[7:0] - resv  Src3[0] - resv  Src4[0] - destReg | 0011 | This instruction executes the RELU activation function on the neurons, then stores the resulting neuron vector in the Local Register, as indicated by the Src4 in the instruction |
| sigmoid\_store  Src1[7:0] - shiftCount  Src2[7:0] - resv  Src3[0] - resv  Src4[0] - destReg | 0100 | This instruction executes the SIGMOID activation function on the neurons, then stores the resulting neuron vector in the Local Register, as indicated by the Src4 in the instruction |
| SetNeuronCount n  Src1[7:0] - n  Src2[7:0] - resv  Src3[0] - resv  Src4[0] - resv | 0101 | This instruction sets the internal neuron count register to the value of src1. When a program starts the neuron count is defaulted to 128. The program should set the neuron count to less than 64 prior to the relu\_prefix or sigmoid\_prefix instruction. |
| ReLU\_prefix  Src1[7:0] - shiftCount  Src2[7:0] - threshold  Src3[0] - resv  Src4[0] - resv | 0110 | This instruction executes the RELU activation function on the neurons, then sorts the resulting neuron vector to find the numerically largest neuron. If this value is greater than the threshold (src2) then the position of this neuron is written out as the final prefix number. If the largest neuron is **not** greater than the threshold (src2) then 0 is written out as the final prefix number.  Note1 : the neuron count register holds the total number of neurons to sort. Since the maximum prefix number is 63 the neuron count should not be set greater than 64 when this instruction is executed.  Note2 : the threshold value (src2) must be specified as an unsigned number (0 to 255).  Note3: A prefix of 0 can result from either the maximum neuron not exceeding the threshold or if the neuron at index 0 is the largest neuron value. |
| sigmoid\_prefix  Src1[7:0] - shiftCount  Src2[7:0] - threshold  Src3[0] - resv  Src4[0] - resv | 0111 | This instruction executes the Sigmoid activation function on the neurons, then sorts the resulting neuron vector to find the numerically largest neuron. If this value is greater than the threshold (src2) then the position of this neuron is written out as the final prefix number. If the largest neuron is **not** greater than the threshold (src2) then 0 is written out as the final prefix number.  Note1 : the neuron count register holds the total number of neurons to sort. Since the maximum prefix number is 63 the neuron count should not be set greater than 64 when this instruction is executed.  Note2 : the threshold value (src2) must be specified as a signed number (-128 to 127).  Note3: A prefix of 0 can result from either the maximum neuron not exceeding the threshold or if the neuron at index 0 is the largest neuron value. |
| Load\_accumulator  Src1[7:0] = cptr  Src2[7:0] = resv  Src3[0] = resv  Src4[0] = resv | 1000 | This instruction loads the accumulator with bias values from coefficient memory for each neuron using src1 as the index into the coefficient memory.  acc = CoeffMem[src1]; |
| Mac  Src1[7:0] - cptr  Src2[7:0] - aptr  Src3[0] - Input/Local  Src4[0] - high/low | 1001 | This instruction performs one mac operation for each neuron, saturating, if the result exceeds 20- bits. Note: 4 instructions (i.e. NOP) are needed between an instruction that sores a value to the Local Register and this instruction which uses that value. Note: the Input Registers (IR0/IR1) contain unsigned values, and the Local Registers contain signed values. The hardware performs sign extensions as needed and the multiplication is always a 9 bits signed operation (not shown below). Refer to the ALU description, paragraph 2.3.11 for details.  rx <= SrcAddrTable[src2];  case({src3,src4})  00: datareg = IR0;  01: datareg = IR1;  10: datareg = LR0;  11: datareg = LR1;  endcase  for(i=0;i<`N\_PREFIX\_NEURONS;i=i+1) begin  neuron\_in[i] <= datareg[rx[i]];  end  acc = (CoeffMem[src1] \* neuron\_in) + acc; |
| HOLD | 1010 | This instruction is similar to the HALT instruction except it holds the pipeline at its current state. The HALT instruction will march the NOP down the pipeline and clear the contents of the pipeline registers. This instruction is useful for Design Verification and/or debugging purposes. |

### Pipeline Block Diagram



Figure Pipeline Block Diagram

### Program Counter

The Program Counter (PC) is an 8 bit counter which holds the address of the current instruction. The PC is reset to 0, and each program begins at PC=0. As each instruction is fetched from the instruction Memory the PC is incremented by 1. The instruction set contains no branch instructions. A new instruction is fetched on each clock cycle. NOP instructions must be included in the program to handle pipeline sequencing. A HALT instruction will cause the PC to be set to zero (PC=0). Once halted the PC will not increment until the Feature Extractor indicates the Input Registers have been loaded with new data. Note: a HALT instruction performs an implicit UNLOCK instruction.

### Instruction Memory

The Instruction memory is sized to hold up to 256 24-bit instructions. These instructions have to be loaded when the processor is in the halt state. While the architecture is flexible, in the final system, there will be limited number of program sequences. One instruction is fetched from memory each cycle and sent down the instruction pipeline.

### Input Registers

There are two 1024-bit input registers (IP1,IP0). The input registers are loaded with the feature counter values that were written into the feature counter FIFO by the feature extractor. The Feature Extractor can continue to work on the next frame data as long as this FIFO is not full. The uSequencer can read the input register, but not write to them. The Input Registers have an unlock/lock behavior as follows:

* If the feature counter FIFO’s are not empty and the input registers are unlocked then data is transferred to the Input Registers. Once the data is written into the Input Registers, they are locked.
* When locked, they cannot be overwritten with new feature counter data from the feature counter FIFO.
* Once the data is consumed, the recognizer can unlock the registers. This is accomplished by executing either the UNLOCK , HALT or HOLD instruction.

Note that since every Recognizer program ends with a HALT/HOLD instruction, which unlocks the Input Registers, the UNLOCK instruction is not required. The Recognizer program can issue the UNLOCK instruction when it is done using the data in the Input Registers. Using the UNLOCK instruction in this manner would only have a small performance benefit as the HALT/HOLD instruction may not have to wait for the next set of features to be loaded into the Input Registers.

### Local Registers

There are two 1024-bit registers (LR1/LR0) to store the output of neurons. These registers can be read and written by the uSequencer.

### Source Address Table Memory

The Source Address Table Memory is 128 entries deep. Each entry can store 128 7-bit unsigned numbers. There can be up to 128 inputs (which can be output of previous layer neurons) in each layer of neural network. Any one of these 128 inputs can feed a neuron. An entry in the SrcAddrTable specifies which of these inputs feeds each of the neurons in the execution of an instruction. For example, entry in SrcAddrTable [i][j] specifies which input feeds the jth neuron during the execution of an instruction with “i” specified as SrcAddrTable entry. . Instructions which utilize the Source Address Table data utilize src1 as the pointer (aptr) into the Source Address Table Memory as follows:

rx <= SrcAddrTable[src2];

Suppose a layer has sixteen inputs for each neuron. Then we need sixteen SrcAddrTable entries to specify neuron sources for each mac operation. To support 32 inputs in input layer and sixteen inputs in about four hidden layers and an output layer, for memory sizing purposes, 128 entries are provided.

### Crossbar unit (xbar)

A flexible crossbar unit is provided to feed one input to each of the 128 neurons from 128 possible inputs. If the one of the Local Registers is selected then a flag is set to indicate that the resulting neuron will be a signed value.

Pseudo code for implementing XBAR is as follows:

rx <= SrcAddrTable[src2];

neuron\_signed <= src3;

case({src3,src4})

00: datareg = IR0;

01: datareg = IR1;

10: datareg = LR0;

11: datareg = LR1;

endcase

for(i=0;i<`N\_PREFIX\_NEURONS;i=i+1) begin

neuron\_in[i] <= datareg[rx[i]];

end

### Coefficient Table Memory

The Coefficient Table Memory is sized to hold 128 sets of weights. Each entry in the Coefficient Table Memory can hold 128 weights (one for each neuron). Each coefficient is an 8-bit 2’s complement number. Each layer, assuming 16 inputs per neuron, will need seventeen sets of coefficients (this includes bias information). Coefficient data has to be loaded during initialization when the recognizer is in halt state. Instructions which utilize the Coefficient data utilize src1 as the pointer (cptr) into the Coefficient Table Memory as follows:

Coefficient Sel:

coeff = CoeffMem[src1];

### ALU

The ALU is a 128 way SIMD vector processing unit. The alu performs lane arithmetic, with 128 lanes, where the result of each lane is a 20 bit value. The result of the operation is stored in the accumulator (acc), which is organized as 128x20 bit (2560 bits) wide result register.

The ALU will perform the following operations based on the instruction operation type.

Where: N\_PREFIX\_NEURONS = 128

#### Mac

Each lane performs a multiply-accumulate operation using the 8-bit signed input coefficient and the 8-bit neuron\_in, which may be signed or unsigned. The result is stored as a 20 bit signed result in the accumulator. If the result exceeds the 20 bit value (i.e. carry = true), then the result saturates at 20 bits, preserving the sign (+524287 or -524288). Since feature match counts are always positive, and it is useful to span the entire range from 0 to 255, they are always treated as an unsigned operation. The Local register may contain the results of a Sigmoid operation, which can be negative, and so must be treated as a signed operand. This may require sign extension. Given these requirements, the multiplication is done as 9 bits.

// coefficient is 8 bit signed number so sign extend into 9 bit signed num

rec\_alu\_opa = {rec\_di\_coeff2[7],rec\_di\_coeff2[7:0]};

// if neuron is signed, then sign extend, else convert to 9 bit pos number

if(neuron\_signed) begin

rec\_alu\_opb = {rec\_di\_neuron2[7],rec\_di\_neuron2[7:0]};

end

else begin

rec\_alu\_opb = {1'b0,rec\_di\_neuron2};

end

for(i=0;i<`N\_PREFIX\_NEURONS;i=i+1) begin

acc[i] <= acc[i] + (rec\_alu\_opa[i] \* rec\_alu\_opb[i]);

end

#### Load

Each lane overwrites the existing acc with the value in the src1 (coeff) input.

Coefficient Sel:

coeff = CoeffMem[src1];

ALU operation:

for(i=0;i<`N\_PREFIX\_NEURONS;i=i+1) begin

acc[i] <= {{13{coeff[i][7]}},coeff[i][6:0]}; //sign extension

end

### Activation Function

The activation function operates on the acc, and is selected by the operation\_type to be either a Sigmoid or ReLU (rectified linear unit) Function. Optionally the activation result can be stored in one of the local registers (LR1/LR0).

#### Sigmoid

The Sigmoid activation function first performs a shift operation on the existing acc, saturating at 12 bits, preserving the existing sign bit(acc[11]). The number of bit positions to shift is determined by the coeff input, src1 in the sigmoid\_store or sigmoid\_prefix instruction. The maximum shift count is 8. The result will be a 12 bit signed value. The pseudo code is shown below:

//--------------------------------------------------------------------------

// function to shift right by given amount, saturate at 12 bits,

// preserve sign bit

//--------------------------------------------------------------------------

function signed [11:0] ShiftSat12;

input signed [19:0] alu\_acc;

input [3:0] shift\_by;

logic signed [19:0] shifted;

begin:ss12

shifted = alu\_acc >>> shift\_by; // arithmatic shift

// shifted < -2048

if(shifted[19]) begin

ShiftSat12 = ~&shifted[18:11] ? 12'h800 : shifted[11:0] ;

end

// shifted > 2047

else begin

ShiftSat12 = |shifted[18:11] ? 12'h7ff : shifted[11:0];

end

end

endfunction // case

The absolute value of the ShiftSat12 function is then feed into the Sigmoid activation function. The pseudo code is shown below:

//-----------------------------------------------------------------------------

// Sigmoid activation function

//-----------------------------------------------------------------------------

function logic [7:0] SIGMOID\_ACT;

input [11:0] abs\_value;

begin:sigmoid\_func

if(abs\_value < 32) begin

SIGMOID\_ACT = abs\_value[7:0];

end

else if(abs\_value < 64) begin

SIGMOID\_ACT = abs\_value[8:1] + 8'h10;

end

else if(abs\_value < 128) begin

SIGMOID\_ACT = abs\_value[9:2] + 8'h20;

end

else if(abs\_value < 256) begin

SIGMOID\_ACT = abs\_value[10:3] + 8'h30;

end

else if(abs\_value < 512) begin

SIGMOID\_ACT = abs\_value[11:4] + 8'h40;

end

else if(abs\_value < 1024) begin

SIGMOID\_ACT = {1'b0,abs\_value[11:5]} + 8'h50;

end

else begin

SIGMOID\_ACT = {2'b00,abs\_value[11:6]} + 8'h60;

end

end // block: sigmoid\_func

endfunction

Since the Sigmoid function operates on the absolute value of the acc, the result must be converted back to a signed number, preserving the original sign bit.

for(i=0;i<`N\_PREFIX\_NEURONS;i=i+1) begin

sigmoid\_result[i] <= SIGMOID\_ACT(rec\_alu\_acc[i]);

if(orig\_sign\_bit) begin

act\_result[i] <=(~sgimoid\_result[i] + 1);

else begin

act\_result[i] <=[ sgimoid\_result[i]);

end

end

#### Relu

The Relu function first performs a shift operation on the existing acc, saturating at 8 bits, preserving the existing sign bit(acc[7]). The number of bit positions to shift is determined by the op\_a (coeff) input. The maximum shift count is 8. The result will be an 8 bit signed value.

//-----------------------------------------------------------------------------

// function to shift right by given amount, saturate at 8 bits,

// preserve sign bit

//-----------------------------------------------------------------------------

function signed [7:0] ShiftSat8;

input signed [19:0] alu\_acc;

input [7:0] shift\_by;

logic signed [19:0] shifted;

begin:ss8

shifted = alu\_acc >>> shift\_by; // arithmatic shift

// shifted < -128

if(alu\_acc[19]) begin

ShiftSat8 = ~&shifted[18:7] ? 8'h80 : shifted[7:0] ;

end

// shifted > 127

else begin

ShiftSat8 = |shifted[18:7] ? 8'h7f : shifted[7:0];

end

end // block: ss8

endfunction // case

The result value of the ShiftSat8 function is then feed into the Sigmoid activation function. The pseudo code is shown below:

//-----------------------------------------------------------------------------

// Relu activation function

//-----------------------------------------------------------------------------

function logic [7:0] RELU\_ACT;

input signed [7:0] alu\_acc;

begin:relu\_func

if (alu\_acc < 0) begin

RELU\_ACT = 8'd0;

end

else begin

RELU\_ACT = alu\_acc[7:0];

end

end

endfunction

### Neuron Sort

The Neuron Sort function finds the numerically largest neuron, using a multistage sort. The Neuron Sort operates on the output of the activation result, which is a register with 128 8-bit signed values. When the Neuron Sort instruction is executed it will find the Neuron with the largest value subject to the current value of “Neuron Count”. Neuron Count must be programmed to a value less than or equal to 64 to restrict the output to 0-63 so that the result can be the Prefix Selector which is constrained to 64. However the neuron count register holds the total number of neurons to sort The position of the largest neuron will be the selected Prefix, an unsigned number from 0 to 63. If the maximum neuron value is not greater than the threshold, given by the instruction, then the prefix number is set to 0;

The pseudo code is shown below:

max\_neuron = act\_result[0];

for(i=0;i<neuron\_cnt;i=i+1) begin

if(act\_result[i] > max\_neuron) begin

max\_neuron = act\_result[i];

max\_index = i[5:0];

end

end

if(max\_neuron > threshold)

prefix\_num = max\_index;

else

prefix\_num = 0;

### Prefix Store

The selected Prefix is stored in the output FIFO, when there is space available. Normally the program will complete execution with a HALT instruction after the prefix is stored.

### Pipeline stalls

The uSequencer will not move from the HALT state if any of the following conditions is true:

* If the uSequencer is not enabled via the config registers. Note at reset time the uSequencer will not be enabled. This allows loading of all the config data while the uSequencer is in a safe state.
* If the uSequencer is enabled, PC=0 and the Input Registers (IR) are empty, i.e. feature counts not available. The pipeline will stall waiting for feature counts to be loaded by the Feature Extractor.
* If the Outbound FIFO does not have space available for at least one prefix.

As a result pipeline stall logic will not be required in this design. Once the uSequencer begins execution of a program it will continue uninterrupted until the HALT instruction is executed. Each program must write only one prefix value to the output FIFO.

### Errors

A single Error Status register will be provided as read-only. Critical pipeline errors will be mapped to individual bits of this register. All internal memories will have ECC error detection which can generate errors and interrupts, if uncorrectable errors occur in a memory read operation. These include:

* Instruction Memory (256x24)
* Coefficient Table Memory (128x1024)
* Source Address Table Memory (128x1024)

# Debug

## Statistics

The Prefix engine will maintain counters which can be read via the APB interface. All counters are set to zero at reset time and can be set to zero via a CSR register access via the APB interface.

The following counters shall be included:

1. Number of Incoming Frames i.e. SOF received
2. Number of Outgoing Frames i.e. SOF sent
3. Number of Incoming Frames with errors
4. Incoming bytes
5. Outgoing bytes
6. Count for each prefix number selected
7. Count of uSequencer Errors detected

## Control Status Registers (CSR)

The Prefix engine will include Control Status Registers (CSR) accessible via the Rbus interface to facilitate Debug/Monitor of the Prefix Engine. These CSR’s shall include the following:

1. Incoming FIFO Status, Flags, Depth, etc
2. Outgoing FIFO Status, Flags, Depth, etc
3. Feature Counters
4. Feature Unlock status
5. uSequencer Program Status Register (PSR)

## uSequencer Debug Mode

The Prefix Engine will include the ability to put the uSequencer in a debug mode, halt the execution and examine the contents of pipeline registers. This functionality will be provided via Rbus accessible CSR’s. The following will only be available when in Debug mode and will have no effect otherwise.

* Breakpoint – A breakpoint address register and control will be provided. When enabled the uSequencer will halt the pipeline when the PC=breakpoint address.
* Step Command – A write to the step command address will cause the halted uSequencer PC to increment by 1.
* Run Command – A write to the run command address will cause the halted uSequencer to continue execution at the current PC. The uSequencer will run until a breakpoint or a HALT instruction is reached

The following Registers will be available for read/write access, while in Single Step Mode:

1. PC
2. Input Registers
3. Local Registers
4. Accumulator
5. Activation Result Register
6. Sort Result Registers
7. Prefix output Register

# Appendix

## Feature Extractor Psuedo code

Reference C++ code that implements this:

#include <stdio.h>

#include <ctype.h>

// Feature extraction control register defines

//

// bit <3> - Prior Condition Control

// Enable previous match to affect current match. I.e. previous feature

// must match, and current feature must match, to assert match output

#define USE\_PRIOR\_FEATURE (1 << 3)

// bit <2> - No Delay Match Output

// Deaserted to delay results by one clock. Implies current match output

// is from previous character. Use for string matching

// Asserted to output results immediately. Implies current match output

// is from current character. Use for range/set matching.

#define NO\_DELAY\_MATCH (1 << 2)

// bits <1:0> - Feature compare command

// Assert match output if feature test returns true

#define FEAT\_CMP\_MASK 0x3

#define FEAT\_EQ 0

// Assert if feature greater than or equal to

#define FEAT\_GT\_EQ 1

// Assert if feature less than

#define FEAT\_LT 2

// Assert if equal, or previous asserted

#define FEAT\_PASS\_EQ 3

// Control register setups

#define NOP (FEAT\_LT)

// For string matching. E.g. BEEF

// String matches if successive chars match 2 to 4 chars

#define MATCH\_START (FEAT\_EQ)

#define MATCH\_CONT (USE\_PRIOR\_FEATURE | FEAT\_EQ)

// Range matching. E.g. A < char < Z

// Note: ranges must compare incoming against both low and high

#define RANGE\_START (FEAT\_GT\_EQ | NO\_DELAY\_MATCH)

#define RANGE\_CONT (USE\_PRIOR\_FEATURE | FEAT\_LT | NO\_DELAY\_MATCH)

// Set matching: char == A or char == 4 or char == S or char == P

// Character sets also compare incoming char

// Set count increases if character matches one of 2 to 4 chars

#define SET\_START (FEAT\_EQ | NO\_DELAY\_MATCH)

#define SET\_CONT (USE\_PRIOR\_FEATURE | FEAT\_PASS\_EQ | NO\_DELAY\_MATCH)

// Hardware defines features in groups of four, with the last feature match

// providing the feature counter increment

#define FEAT\_MATCH\_1\_CHAR { 0, 0, 0, FEAT\_EQ }

#define FEAT\_MATCH\_2\_CHAR { 0, 0, MATCH\_START, MATCH\_CONT }

#define FEAT\_MATCH\_3\_CHAR { 0, MATCH\_START, MATCH\_CONT, MATCH\_CONT }

#define FEAT\_MATCH\_4\_CHAR { MATCH\_START, MATCH\_CONT, MATCH\_CONT, MATCH\_CONT }

// Does not include ending range character

#define FEAT\_RANGE { 0, 0, RANGE\_START, RANGE\_CONT }

// Includes ending range character

#define FEAT\_RANGE\_INCL { 0, RANGE\_START, RANGE\_CONT, SET\_CONT }

#define FEAT\_SET\_1\_CHAR FEAT\_MATCH\_1\_CHAR

#define FEAT\_SET\_2\_CHAR { 0, 0, SET\_START, SET\_CONT }

#define FEAT\_SET\_3\_CHAR { 0, SET\_START, SET\_CONT, SET\_CONT }

#define FEAT\_SET\_4\_CHAR { SET\_START, SET\_CONT, SET\_CONT, SET\_CONT }

class CMatchElement {

private:

unsigned char m\_Count; // one counter per group of four features

unsigned char m\_Control[4]; // Feature control registers for this group

unsigned char m\_Feature[4]; // Feature match values

unsigned char m\_History[4]; // Save previous characters [debugging]

unsigned int m\_CharCount; // Input character count

unsigned int m\_Debug;

bool m\_PriorMatch[4]; // Feature matches, current char

bool m\_PriorDelay[4]; // Feature matches, prev char

public:

// Set debug state

void SetDebug(unsigned int debug) { m\_Debug = debug; }

// initialize count and internal state

void InitCount() {

for (int i = 0; i < 4; i++) {

m\_PriorMatch[i] = false;

m\_PriorDelay[i] = false;

m\_History[i] = 0;

}

m\_CharCount = 0;

m\_Count = 0;

};

// Get count value

unsigned int GetCount() { return m\_Count; };

void PutControl(unsigned char \*control) {

for (int i = 0; i < 4; i++) {

m\_Control[i] = control[i];

}

};

void PutMatch(unsigned char \*match) {

for (int i = 0; i < 4; i++) {

m\_Feature[i] = match[i];

}

};

// Do single character feature extraction process

bool Process(unsigned char input)

{

bool result = false;

// First, copy previous results to delay results

for (int j = 0; j < 4; j++) m\_PriorDelay[j] = m\_PriorMatch[j];

// Save input history

for (int j = 0; j < 3; j++) m\_History[j] = m\_History[j+1];

m\_History[3] = input;

// Do match detection

for (int j = 0; j < 4; j++) {

// (Pass through for FEAT\_PASS\_EQ case is done below)

switch (m\_Control[j] & FEAT\_CMP\_MASK) {

case FEAT\_EQ: result = (input == m\_Feature[j]); break;

case FEAT\_GT\_EQ: result = (input >= m\_Feature[j]); break;

case FEAT\_LT: result = (input < m\_Feature[j]); break;

case FEAT\_PASS\_EQ: result = (input == m\_Feature[j]); break;

}

// Do previous match handling

// Note that feature zero can not use prior matches

bool prior;

if ((j > 0) && (m\_Control[j] & USE\_PRIOR\_FEATURE)) {

// Generate prior feature output

// If no delay == 0, then use prev char, previous feature

// If no delay == 1, then use current char, previous feature

prior = (m\_Control[j-1] & NO\_DELAY\_MATCH) ? m\_PriorMatch[j-1] : m\_PriorDelay[j-1];

// accumulate match

if ((m\_Control[j] & FEAT\_CMP\_MASK) == FEAT\_PASS\_EQ) {

// Pass through case (range and set matching)

result = result | prior;

} else {

// String matching

result = result & prior;

}

}

if (result && m\_Debug) {

printf("i/prior/Char/feat: %d %d %02x %02x\n",

j, m\_PriorMatch[j], input, m\_Feature[j]);

}

// Update match results

m\_PriorMatch[j] = result;

}

// Update feature count

// Saturate at 255

m\_Count = (result && (m\_Count < 255)) ? m\_Count + 1 : m\_Count;

if (result && m\_Debug) {

printf("Match/Count/Hist: %3d %7d ", m\_Count, m\_CharCount);

for (int j = 0; j < 4; j++) {

if (isalnum(m\_History[j])) printf("%c", m\_History[j]);

else printf("\*");

}

printf("\n");

}

m\_CharCount++;

};

};

int main(int argc, char \*argv[]) {

FILE \*input;

char currChar;

CMatchElement matcher;

unsigned char control[10][4] = {FEAT\_MATCH\_1\_CHAR,

FEAT\_MATCH\_2\_CHAR,

FEAT\_MATCH\_3\_CHAR,

FEAT\_MATCH\_4\_CHAR,

FEAT\_RANGE,

FEAT\_RANGE\_INCL,

FEAT\_SET\_1\_CHAR,

FEAT\_SET\_2\_CHAR,

FEAT\_SET\_3\_CHAR,

FEAT\_SET\_4\_CHAR};

unsigned char match[10][4] = {{ 0, 0, 0, 'P'},

{ 0, 0, 'A', 'P'},

{ 0, 'H', 'A', 'P'},

{'C', 'H', 'A', 'P'},

{ 0, 0, '0', '9' }, // End non inclusive

{ 0, '0', '9', '9'}, // End is included

{ 0, 0, 0, 'P'},

{ 0, 0, 'Z', 'P'},

{ 0, 'Q', 'Z', 'P'},

{'G', 'Q', 'Z', 'P'}};

if ((input = fopen(argv[1], "r")) == NULL) {

printf("Unable to open: %s\n", argv[1]);

return -1;

}

for (int i = 0; i < 10; i++) {

matcher.InitCount();

matcher.PutControl(control[i]);

matcher.PutMatch(match[i]);

//matcher.SetDebug(i == 4);

/\*

printf("Control: ");

for (int j = 0; j < 4; j++) {

printf("%02x ", (int)control[i][j]);

}

printf("\n");

\*/

while((currChar = fgetc(input)) != EOF) {

matcher.Process(currChar);

}

printf("case: %d count: %d\n", i, matcher.GetCount());

rewind(input);

}

return 0;

}

/\* Sample output:

./feature\_match alice29.txt

case: 0 count: 64

case: 1 count: 12

case: 2 count: 12

case: 3 count: 12

case: 4 count: 1

case: 5 count: 2

case: 6 count: 64

case: 7 count: 65

case: 8 count: 149

case: 9 count: 231

Analysis using grep:

case 0, single char match of 'P':

grep -b -o P alice29.txt | wc -l

64

case 1: char match of string 'AP':

grep -b -o AP alice29.txt | wc -l

12

case 2: char match of string 'HAP':

grep -b -o HAP alice29.txt | wc -l

12

case 3: char match of string 'CHAP':

grep -b -o CHAP alice29.txt | wc -l

12

case 4, 5: range match 0 to 9:

grep -b -o 0 alice29.txt | wc -l

0

grep -b -o 1 alice29.txt | wc -l

0

grep -b -o 2 alice29.txt | wc -l

1

grep -b -o 3 alice29.txt | wc -l

0

grep -b -o 4 alice29.txt | wc -l

0

grep -b -o 5 alice29.txt | wc -l

0

grep -b -o 6 alice29.txt | wc -l

0

grep -b -o 7 alice29.txt | wc -l

0

grep -b -o 8 alice29.txt | wc -l

0

grep -b -o 9 alice29.txt | wc -l

1

Results:

case 4: 0 to 9, end range exclusive: 1

case 5: 0 to 9, end range inclusive: 2

Set matching:

grep -b -o P alice29.txt | wc -l

64

grep -b -o Z alice29.txt | wc -l

1

grep -b -o Q alice29.txt | wc -l

84

grep -b -o G alice29.txt | wc -l

82

Results:

case 6: 'P' : 64

case 7: 'P' + 'Z' : 64 + 1 = 65

case 8: 'P' + 'Z' + 'Q': 64 + 1 + 84 = 149

case 9: 'P' + 'Z' + 'Q' + 'G': 64 + 1 + 84 + 82 = 231

\*/

## Sigmoid Activation Function

Sigmoid activation function to be implemented is a simplified version of the actual sigmoid function. A pseudo-code for this function is as follows:

int8 SigmoidFn(int12 input)

{

sign = 0;

absval = input;

if (input < 0)

{

Absval = -input;

Sign = 1;

}

switch (absval)

{

Case 0 <= absval < 32:

Out = absval;

Break;

Case 32 <= absval < 64:

Out = (absval >> 1) + 16;

Break;

Case 64 <= absval < 128:

Out = (absval >> 2) + 32;

Break;

Case 128 <= absval < 256:

Out = (absval >> 3) + 48;

Break;

Case 256 <= absval < 512:

Out = (absval >> 4) + 64;

Break;

Case 512 <= absval < 1024:

Out = (absval >> 5) + 80;

Break;

Case 1024 <= absval < 2048:

Out = (absval >> 6) + 96;

Break;

default:

// it should not come here

break;

}

If (sign)

Out = -Out;

return Out;

}

## Program Example

For the purpose of illustration, pseudo code for implementing a simple feature extraction program is given in Table 1. The following parameters are assumed here

* Number of inputs = 256.
* Number of NN layers = 3.
* Number of inputs per neuron in first layer = 4, with two coming from first 128 inputs and the third coming from the last 128 inputs.
* Number of inputs per neuron in second layer = 3.
* Number of neurons per neuron in output layer = 2.
* Number of neurons in first two layers = 128.
* Number of neurons in output layer = 64.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PMEM Location** | **Opcode** | **Coeff Table Addr** | **Addr Table Input** | **Source Register** | **Explanation** |
| **Specify neuron count** | | | |  |  |
| 0 | SetNeuronCount | 128 |  |  | Neuron Count = 128 (this sets SIMD width) - we can use it for clock level gating of macs for low power application |
| **Load bias data** |  |  |  |  |  |
| 1 | Load Acc | coeff[0] |  |  | Load Bias to accumulators from coeff table[0] |
| **2 macs for each neuron from first 1024-bit data** | | | |  |  |
| 2 | mac | coeff[1] | atable[0] | IR[0] | Perform MAC operation. Weights to be read from coeff table [1] Data comes from first word of feature extractor data The input data is shuffled by crossbar using mapping in address table [0].  Note: actual data register accessed depends on which feature vector output is being processed. |
| **Second Layer - Input is from local register.** | | | | |  |



Figure One input per neuron

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | mac | coeff[2] | atable[1] | IR[0] |  |



Figure Two inputs per neuron

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2 macs for each neuron from second 1024-bit data** | | | |  |  |
| 4 | mac | coeff[3] | atable[2] | IR[1] | Perform next set of macs. Same as before, but we now use second input register |
| 5 | mac | coeff[4] | atable[3] | IR[1] |  |



Figure Four inputs per neuron

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Now One layer computation is done** | | | |  | |  | |
| relu\_and\_st | 3 |  | LR[0] | | Perform ReLU operation and store  Mac output is shifted right by 3-bits prior to ReLu operation. | |
| unlock\_ireg |  |  |  | | This instruction frees the input register so that the next set of data from feature vector can be loaded into input register while processing is still going on. | |
| nop |  |  |  | | Data stored in register is not available for use. Need three nops between st and mac. | |
| **Second Layer - Input is from local register.** | | | | | |  | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Start Second Layer compute** | | | | |  |
| **Load bias data** |  |  |  |  |  |
| 9 | Load Acc | coeff[5] |  |  | Load Bias to accumulators from coeff table[5] |
| 10 | mac | coeff[6] | atable[4] | LR[0] | Perform MAC operation. Weights to be read from coeff table [6] Data comes from local register The input data is shuffled by crossbar using mapping in address table [4]. |
| 11 | mac | coeff[7] | atable[5] | LR[0] |  |
| 12 | mac | coeff[8] | atable[6] |  |  |
| 13 | relu\_and\_st | 3 |  | LR[0] | Perform ReLU operation and store  MAC output is shifted right by 3 bits prior to ReLU operation. |
| 14 | nop |  |  |  |  |

3 inputs per neuron on the second level



Figure Three inputs per neuron, second level

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Final Layer** |  |  |  |  |  |
| 15 | SetNeuronCount | 64 |  |  | Output layer has 64 neurons |
| 16 | Load Acc |  | coeff[9] |  | Load bias |
| 17 | mac | atable[7] | coeff[10] | LR[0] |  |
| 18 | mac | atable[8] | coeff[11] | LR[0] |  |
| 19 | relu\_and\_GetPrefixNum | 3 |  | LR[0] | Perform Relu and then get prefix number. This instruction executes in multi-cycles.  Mac output is shifted right by 3 bits prior to ReLU operation. |
| 20 | Halt |  |  |  | End of processing. If more data available, the processign starts at PC=0. Otherwise, processor goes to IDLE state. |

General formula for processing time is (assuming 128 or fewer neurons in each layer)

Where:

L = Number of layers in NN

Ni = Number of neuron inputs in layer “i”

For example with 4 layer NN, with 32 inputs per neurons in the first layer and 16 in the other layers, processing time is 98.