doi:10.3772/j.issn.1002-0470.2013.12.013

# 多元 LDPC 译码器的设计与实现①

黎海涛②* 杨磊磊* 刘 飞** 袁海英*

(*北京工业大学电子信息与控制工程学院 北京 100124)

(**中兴通信股份有限公司 北京 100191)

**摘 要** 针对一般多元 LDPC 译码器时延大、吞吐量低的局限,设计了一种基于 EMS 算法的新型多元 LDPC 译码器。它根据前向后向算法规则,以 3 路单步运算单元完成校验节点更新,使得所需时钟周期约降为一般结构的 1/3;采用低复杂度全并行运算的变量节点信息更新单元,无需利用前向后向算法将更新过程分解为多个单步运算,消除了变量节点更新的递归计算;采用新的双进双出信息调度算法,进一步降低了变量节点更新复杂度且提高了译码器吞吐量。通过 Xilinx Virtex-4 平台对一个 GF(16)域上(480,360)的准循环多元 LDPC 码进行了综合仿真,结果表明,它以较小的逻辑资源消耗为代价提高了近 3 倍的吞吐量。

**关键词** 扩展最小和,多元 LDPC 码,FPGA,吞吐量

## 0 引 言

近年来,由二元低密度奇偶校验(low-density parity check,LDPC)码延伸得到的多元 LDPC 编码受到广泛关注,它具有更强的抗突发错误能力,适用于高速率无线传输[1]。多元 LDPC 编码可以采用与二元 LDPC 码类似的准循环(QC)编码算法,通过在 GF($q$)域进行置换运算,把 LDPC 矩阵简化为近似下三角矩阵,大大降低了编码复杂度。多元 LDPC 码的纠错能力随编码阶数的增大而更强。但由于多元 LDPC 是在符号级上译码,标准的和积译码算法的复杂度会随着编码阶数的增加而呈指数级增长,使得专用集成电路(ASIC)或现场可编程门阵列(FPGA)仅能用于低阶编码方案。因此,目前对多元 LDPC 码的研究也大多集中在如何降低其译码器的实现复杂度上。

为了降低译码算法的计算复杂度,文献[2,3]提出了扩展最小和(extended min-sum,EMS)算法,并采用前向后向译码结构将校验/变量节点的更新分解为多个单步运算的时分复用形式。这虽降低了硬件实现的复杂度,但带来了消息向量之间大量的递归计算,导致译码延迟大,故译码器吞吐量受到较大限制。为此本文在文献[4]的基础上,设计了一种基于 EMS 算法的新型多元 LDPC 译码器,该译码器通过提高并行度减少递归运算,采用新的双进双出信息调度算法,进一步降低变量节点更新复杂度且提高了译码器吞吐量。

## 1 EMS 译码算法

多元 LDPC 编码可以通过校验矩阵 $\boldsymbol{H}$ 来实现,文献[4]给出了一种准循环结构的 $\boldsymbol{H}$ 矩阵结构,本文译码器设计以此矩阵为基准。它被划分为 6 行 24 列子矩阵,每个子矩阵是大小为 5 × 5 的零矩阵或经过循环移位等处理的准对角矩阵。

EMS 译码算法的核心思想是在消息传递的过程中只传递 $q$ 维向量中可信度最高的 $N_m$ 个消息值。定义所传递的初始消息向量为 $\boldsymbol{L}$,$\boldsymbol{U}_{tp}$($\boldsymbol{V}_{tp}$)为计算所得变量(校验)节点信息,$\overline{\boldsymbol{U}}_{tp}$($\overline{\boldsymbol{V}}_{tp}$)为所保留的变量(校验)节点信息。整个译码算法可描述为以下步骤:

(1)初始化:定义 $L_k$ 为第 $k$ 个 $q$ 元 LDPC 编码码子对应初始消息值,$m$ 为 $q$ 元 LDPC 编码码子对应

① 国家自然科学基金(61001049),北京市自然科学基金(4112012)和北京市教委科技成果转化(61001049)资助项目。

② 男,1972 年生,博士,副教授;研究方向:无线通信,信号处理等;E-mail:lihaitao@bjut.edu.cn

(收稿日期:2013-03-21)

星座集合中星座个数，$\{y^1,\cdots,y^m\}$ 为接收到用于译码的星座集合，$\{\mu_k^1,\cdots,\mu_k^m\}$ 为 $q$ 元 LDPC 编码第 $k$ 个码字对应调制方式下的星座集合，则初始消息向量计算如下：

$$L_k = -\sum_{i=1}^{m} | y^i - \mu_k^i |^2 \tag{1}$$

(2)度为 $d_v$ 的变量节点信息更新：

$$\boldsymbol{U}_{tp}[i_1,\cdots,i_q] = \overline{\boldsymbol{L}}[i_{r1},\cdots,i_{rN_m}] + \sum_{v=1,v\neq t}^{d_v} \overline{\boldsymbol{V}}_{pv}[i_{s1},\cdots,i_{sN_m}] \tag{2}$$

(3)置换过程：

$$\overline{\boldsymbol{U}}_{pc}[j_1,\cdots,j_{N_m}] = \overline{\boldsymbol{U}}_{vp}[i_1,\cdots,i_{N_m}] \qquad j_x = h\times i_x \tag{3}$$

其中 $x$ 为符号向量 $j$、$i$ 的元素标号，$h$ 为消息向量对应于校验矩阵 $\boldsymbol{H}$ 中的值。

(4)度为 $d_c$ 的校验节点信息更新：

$$\boldsymbol{V}_{tp}[j_1,\cdots,j_q] = \min\Big(\sum_{v=1,v\neq t}^{d_c} \overline{\boldsymbol{U}}_{t,j}[j_1,\cdots,j_{N_m}]\Big) \tag{4}$$

$V_{tp}$ 是 $\{\overline{U}_{pc}\}_{c=1,\cdots,d_c,c\neq t}$ 之间逐次迭代运算，经前向后向算法分解后，其单步运算如下：

$$\boldsymbol{Z}[a_{t1},a_{t2},\cdots,a_{tp}] = \max_{a_r(x)+a_t(x)+a_s(x)=0}\{\boldsymbol{U}_{pr}[a_{r1},a_{r2},\cdots,a_{rp}] + \boldsymbol{U}_{pr}[a_{s1},a_{s2},\cdots,a_{sp}]\}_{r,s=0,\cdots,d_c,r\neq s} \tag{5}$$

(5)逆置换：

$$\overline{\boldsymbol{V}}_{pv}[i_1,\cdots,i_{N_m}] = \overline{\boldsymbol{V}}_{cp}[j_1,\cdots,j_{N_m}], \quad i_x = j_x/h \tag{6}$$

(6)译码输出：

$$\hat{x} = \underset{i\in GF(2^P)}{\operatorname{argmax}}\{\overline{\boldsymbol{L}}[i_1,\cdots,i_{N_m}] + \sum_{v=1}^{d_v}\overline{\boldsymbol{V}}_{pv}[i_1,\cdots,i_{N_m}]\} \tag{7}$$

## 2 译码器设计

### 2.1 整体结构

资源消耗、吞吐量是衡量译码器性能的重要指标，硬件设计时需要在两者之间合理取舍。部分并行译码结构较好地解决了资源与速度的平衡问题。在一般的采用 EMS 算法设计的部分并行译码结构中，为降低硬件资源消耗，采用前向后向算法将校验/变量节点更新分解为多个单步运算，每个校验/变量节点更新运算分配一个单步运算单元[7-9]，具有较小的资源消耗，但译码速度受到较大限制。为此，本文设计的多元 LDPC 译码器采用的部分并行译码结构以 3 路单步运算单元完成校验节点更新，硬件资源消耗有所增加，但所需时钟周期约降为一般结构的 1/3。译码器低复杂度全并行运算的变量节点信息更新单元无需利用前向后向算法将更新过程分解为多个单步运算，消除了变量节点更新时的递归计算。译码器采用双进双出信息调度算法，进一步降低了变量节点更新复杂度，提高了吞吐量。下面详细阐述所设计的多元 LDPC 译码器。

如图 1 所示，该译码器包括控制单元、初始消息计算/存储单元、校验/变量节点更新单元、校验/变量节点信息存储单元、输出缓存单元。控制单元向其他各个单元发出控制信息，确保译码器正常工作。初始消息计算单元完成初始消息计算，并将计算结果存储于对应初始消息存储单元。变量/校验节点更新单元逐次完成校验/变量节点信息之间的迭代交换。迭代更新结束后，输出缓存模块输出译码结果。其中，每一行分块矩阵分配一个校验节点更新单元，每 2 列分块矩阵分配一个变量节点更新单元，以时分复用方式完成校验/变量节点更新。

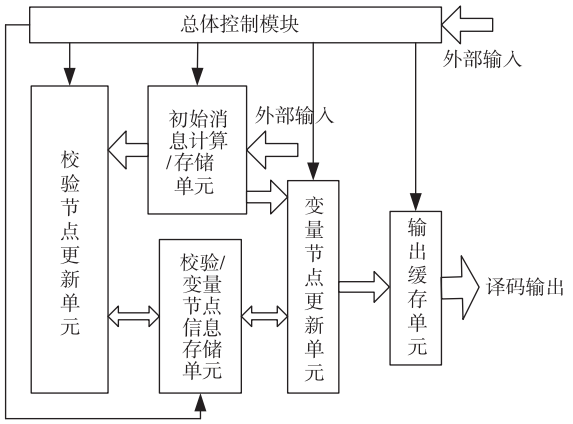


**图 1 整体结构**

在二元 LDPC 码的译码器中，由于变量/校验节点更新周期较短，更新过程中，上次迭代中的更新结果无法擦除，变量/校验节点信息存储在不同的存储单元中。由于多元 LDPC 码变量/校验节点更新周期较长，变量/校验节点信息可存储在相同的存储单元中，以节省存储资源。由于采用双进双出信息调度算法，每一列子矩阵分配 1 个初始消息存储单元，每 1 块非零子矩阵分配 1 个校验/变量节点信息存储单元，分别采用 1 块深度为 80 的双口 RAM 实现，其中低 40 位与高 40 位交替存储来自相邻两帧的消

息向量。图 1 给出的译码器整体结构框图中,初始消息采取 11bit 量化,校验/变量节点信息采取 12bit 量化(包含 4bit 符号信息)。

## 2.2 校验节点更新单元

基于 EMS 算法,文献[5,6]均采用部分并行行译码结构,以 1 路单步运算单元完成一行子矩阵对应校验节点更新(其典型结构如图 2 所示),以最小化芯片面积,译码器吞吐量受到较大限制。

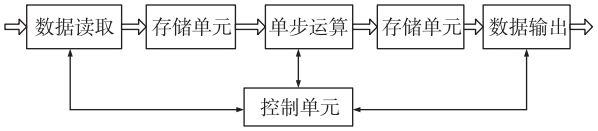


**图 2 校验节点更新结构**

首先,数据读取单元从校验/变量节点信息存储单元中读取本次迭代所需变量节点信息,并完成置换计算,将结果写入存储单元;随后,单步运算单元从存储单元中读取所需信息,经过 $3d-6$($d$ 为校验节点的度)次单步运算逐一完成消息向量更新,并将结果写入存储单元;最后,数据输出单元从存储单元中读取输出结果并作逆置换运算。综合考虑译码器资源消耗和吞吐量,本文提出一种新的译码器结构。它以 3 路单步运算单元完成校验节点信息更新,通过合理分配存储资源,以较少的硬件资源消耗为代价,使得所需时钟周期约降为一般结构的 1/3。

在前向后向算法中,前向与后向更新相互独立,最终节点更新需要来自前向后向更新输出结果。为此,本设计以两路单步运算单元并行完成前向后向更新过程,并以第 3 路单步运算单元完成最终节点更新,在前两路单步运算单元运行时,第 3 路单步运算单元无需等待本行校验节点前向后向更新结束之后再运行。图 3 给出了校验节点更新单元的结构框图,单步运算单元 A、B、C 分别对应图 2 中第一至第三层单步运算。存储单元 A/B 从校验/变量节点信息存储单元中读取本次单步运算所需变量节点信息并完成置换操作(由于单步运算单元 C 计算需要,消息矩阵的第一列与最后一列需同时分别写入存储单元 A/B),根据前向后向算法规则,输出数据至单步运算单元 A/B,单步运算单元 A/B 将计算结果写入存储单元 C(由于最终输出结果还包含前向后向更新部分结果,单步运算单元 A/B 输出结果选择性写入存储单元 D)。在单步运算单元 A,B 完成一行校验节点前向后向更新后,存储器 A/B 开始读取下一行校验节点前向后向更新所需数据。单步运算单元 C 从存储单元 C 读取所需信息,并将结果写入存储单元 D,在单步运算单元 C 计算结束之后,从存储单元 D 中读取数据完成逆置换操作并输出至校验/变量节点信息存储单元。设计中置换/逆置换操作均采用查表法实现,避免有限域的直接运算以降低计算复杂度。

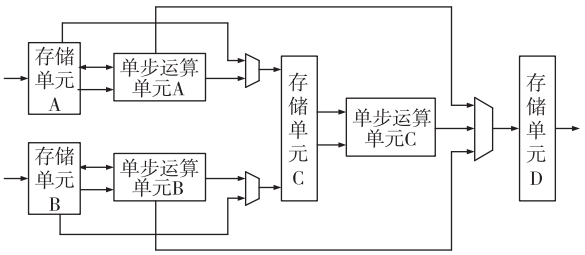


**图 3 提出的校验节点更新结构**

### 2.2.1 存储单元

采用 EMS 译码算法设计的译码器需要用到大量插值排序、重复符号检测等操作,在上述操作结束之后,数据都以向量的形式存在。为减少译码延迟,提高译码吞吐量,本文设计中,数据尽可能采取向量存储(即将一个 $n_m$ 维向量作为一个整体进行读写操作,而不是对向量中元素进行单个操作),若采用单个数据读写操作,反而需要较多的周期完成。但由于单步运算需要遍历两组输入向量中所有元素,部分操作采用向量存储无法完成,为此采用逻辑片搭建数组缓存来自 RAM 输出的向量存储数据。

校验节点更新存储单元 A/B 分别由 3 组深度为 $n_m$ 的基本存储单元构成,用于存储本次单步运算所需消息向量。由于在前向后向计算过程中,第一次递归过程所需信息全部来自变量节点信息,而在其余各次递归过程中,下一次递归计算需要上一次输出作为输入。即存储单元 A/B 需要同时存储来自变量节点信息的数据流和来自单步运算输出的消息向量,因此,存储单元采用逻辑片搭建。

由于变量节点信息采取单个数据逐一读取,需 $n_m$ 个周期完成读操作,而单步运算输出采取向量存储,仅需一个周期。为进一步减少数据读取所需时间,在单步运算过程之中,即读取下一次搜索所需变量节点信息,即在两个存储单元之间完成乒乓操作。图 4 详细描述了本文设计存储单元 A/B 的结构,单步运算单元交替遍历存储单元 1、2。其中第 1、2 个存储单元交替存储来自变量节点的信息;第 3 个存

储单元在第一次递归过程中,存储来自变量节点的信息,其余各次存储来自单步运算输出的消息向量。

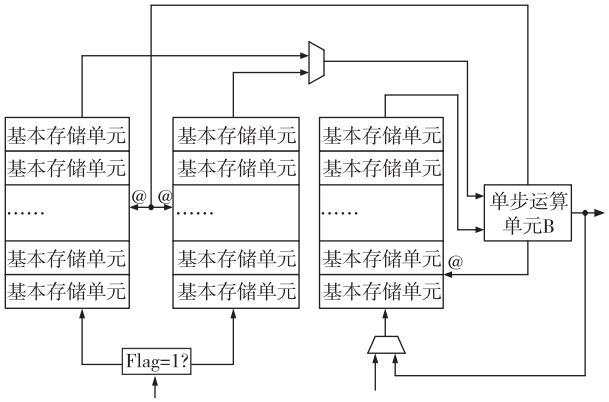


**图 4 存储单元 A/B 结构**

单步运算单元 C 在执行过程中,需要读取单步运算单元 A/B 输出消息向量和来自变量节点的信息。期间单步运算单元 A/B 并没有停止操作等待单步运算单元 C 完成本行校验节点更新结束之后再进行下一行校验节点更新,单步运算单元 C 也无需等待本行校验节点前向后向更新结束之后再进行运算。图 5 详细描述了本文设计存储单元 C 结构。

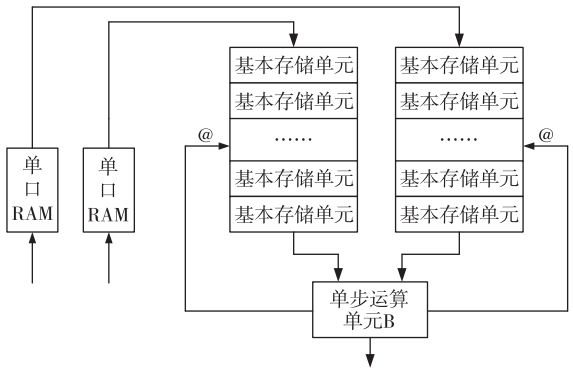


**图 5 存储单元 C 结构**

对度为 8 的校验节点更新,前向后向更新各需 6 次递归计算,通过合理分配读写地址,在前向后向更新完成 3 次递归计算之后,单步运算单元 C 即可开始最终节点更新。结合上述设计,图 6 给出了各个单步运算单元执行的时序关系。提出的校验节点更新单元共需 5.5T 个时钟周期完成本次迭代校验节点更新,约为一般结构的 1/3,其中 $T$ 为一行校验节点前(后)向更新过程所需时间。

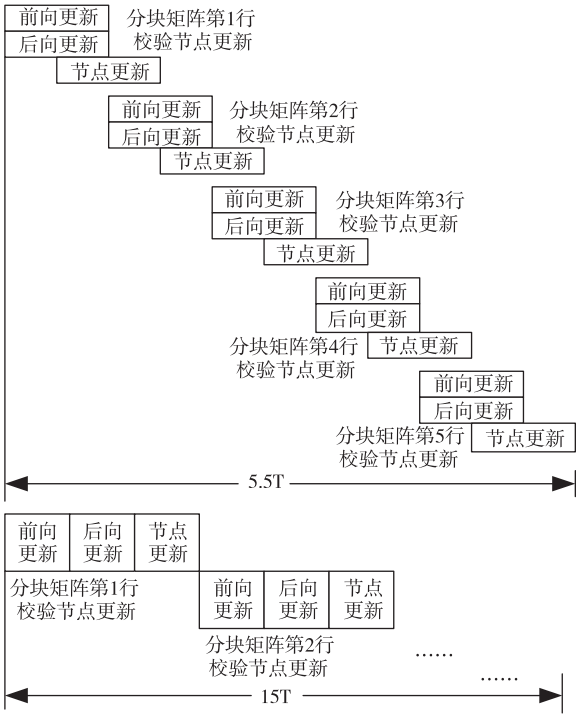


**图 6 单步运算单元的执行顺序**

### 2.2.2 单步运算单元

下面介绍单步运算单元硬件结构,其结构如图 7所示,其中虚线部分对应图 3 中存储单元 A、B、C。定义该模块的两个输入向量为 $\boldsymbol{R}_{\mathrm{A}}$ 、$\boldsymbol{R}_{\mathrm{I}}$ ,对应的

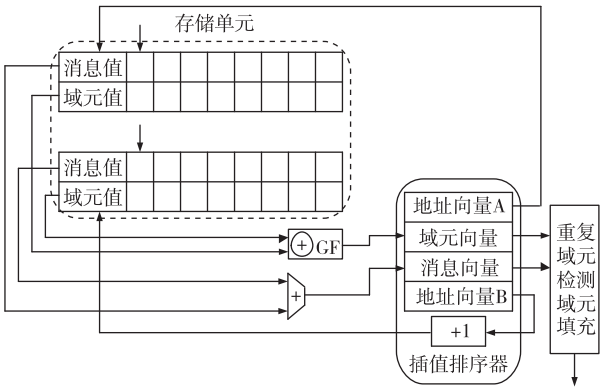


**图 7 校验节点单步运算硬件结构**

符号向量为 $\boldsymbol{\alpha}_{\mathrm{A}}$ 、$\boldsymbol{\alpha}_{\mathrm{I}}$ ,输出向量为 $\boldsymbol{R}_{\mathrm{B}}$ ,对应的符号向量为 $\boldsymbol{\alpha}_{\mathrm{B}}$ 。首先,输入向量 $\boldsymbol{R}_{\mathrm{A}}\backslash\boldsymbol{R}\mathrm{I}$ 写入存储单元 $\boldsymbol{A}\backslash\boldsymbol{I}$,同时,向量 $\boldsymbol{R}_{\mathrm{A}}$ 与向量 $\boldsymbol{R}_{\mathrm{I}}$ 中最大值之和被写入插值排序器(如图 8)用于对插值排序器进行初始化。随后插值排序器每次分别从存储器 A,I 中读取一个数据,将其消息值之和及有限域符号和作为插值排序器输入,并返回一个读地址。具体运算如下:

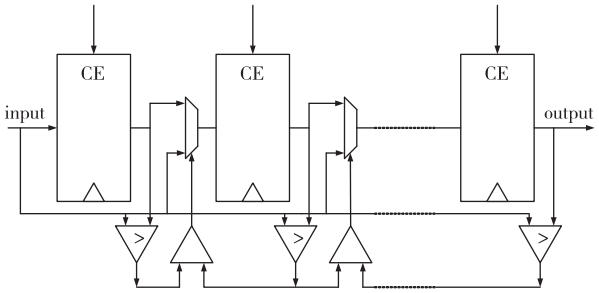


**图 8　插值排序器结构**

```
Initialization:
for all(i from 0 to 7) do
    S⇐{R_A[i] + R_I[0], α_A[i] ⊕ α_I[0], i, 0}
endfor
loop:
for all(i from 0 to 15) do
    i = indexA[0]
    j = indexI[0] + 1
    if(α_A[i] ⊕ α_I[j] ∉ α_B) then
        S⇐{R_A[i] + R_I[j], α_A[i] ⊕ α_I[j], i, j}
    endif
endfor
```

为了消除重复符号以及解决符号填充问题(符号数小于 $n_m$ 时),定义了一个深度为 $n_m$ 的存储器(其值 *Memory. D* 初始化为 0 值,对应符号值 *Memory. S* 分别为 0 到 $n_m$ −1)和一个大小为 $n_m$ 的标记矩阵 FLAG 用于解决此问题。其中插值排序器输出消息值为 *value_in*,其对应符号 *sym_in* 需要分别与存储器中符号进行对比,共有 3 种可能情况:输入符号与存储器中所有符号均不相同;输入符号与存储器中非初始化符号相同;输入符号与存储器中初始化符号相同。描述此方法的伪代码如下:

```
initial: Memory. D = 0; Memory. S = 0; n_m − 1;
    CNT = 0; FLAG = 0;
for i = 1: n_m
  if(sym_in ! = Memory. S)
    CNT = CNT + 1;
    Memory. D[CNT] = value_in;
    Memory. S[CNT] = sym_in;
    FLAG[CNT] = 1;
  elseif((sym_in = = Memory. S[j]&!FLAG[j])
    j ∈ 0: n_m − 1
    CNT = CNT + 1;
    Memory. D[CNT] = value_in;
    Memory. S[CNT] = sym_in;
    Memory. S[j] = Memory. S[CNT];
    FLAG[CNT] = 1;
  elseif((sym_in = = Memory. S[j]&FLAG[j])
    j ∈ 0: n_m − 1
      no operation
    end
  end
```

为保证算法的收敛性,同时使得存储数据尽可能小,以减少数据位宽,需要在每次单步计算结束之后,对消息向量进行归一化处理。采用上述结构,对于度为 9 的校验节点更新完成一次迭代共需 758 个时钟周期,对于度为 8 的校验节点更新完成一次迭代共需 663 个时钟周期。

### 2.3　变量节点更新单元

变量节点更新单元用于实现变量节点更新和译码判决二个功能。文献[5,6]采用前向后向算法将变量节点更新分解为单步运算逐次完成。每个单步运算分为 3 个步骤,需分别逐次遍历输入向量 $\boldsymbol{R}_{\mathrm{I}}$ 和 $\boldsymbol{R}_{\mathrm{A}}$ 中所有符号,找出其中具有相同符号的消息值,并对输出结果进行排序选择归一化处理。

图 9 给出了采用前向后向算法的变量节点更新

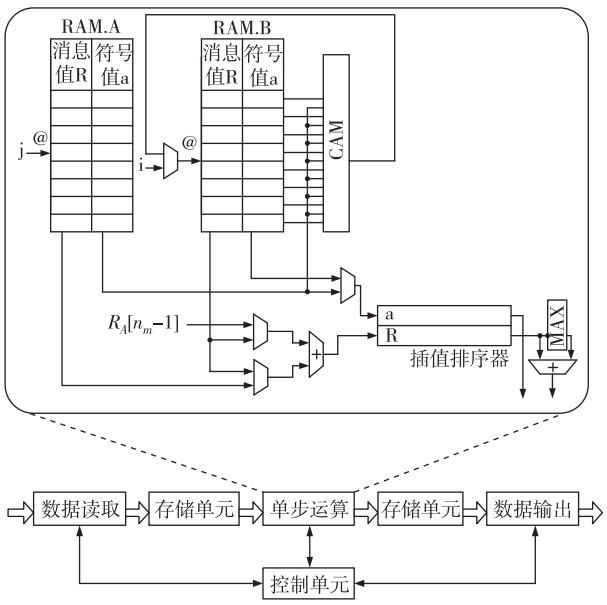


**图 9　变量节点更新结构**

单元一般结构。按照上述方法一次单步运算共需约 41 个时钟周期。对于变量节点度为 3 的译码器而言,共需 5 次单步运算,则完成一次迭代共需 41 ×5 ×5 = 1025 个时钟周期。变量节点更新模块中的单步运算模块的功能分为两个步骤完成,第一个步骤是寻找两个输入向量中域值相同的部分,第二个步骤是处理向量中的剩余部分。具体算法如下:

```
for all(j from 0 to 31) do
    if(a_VI[j] ∈ a_VA) then
```

```
        k:a_VA[k] = a_VI[R_VI[j]
        S⇐{R_VI[j] + R_VA[k],a_VI[j]}
    else
        S⇐{R_VI[j] + R_VA[n_m - 1],a_VI[j]}
    endif
    endfor
    for all(i from 0 to 31) do
        if(a_VA[i] ∉ S)then
        S⇐{R_VA[i] + R_VI[n_m - 1],a_VA[i]}
    endif
    endfor
```

在遍历算法中,具有相同符号的消息值直接相加,具有不同符号的消息值与另一向量中的最小值相加。由于经过归一化处理后的输入向量中最小值为 0,故可以简化此遍历搜索过程,下面具体介绍该方法。

以度为 2 的变量节点信息更新为例,首先,输入消息值 A、B、C(其中 A 为初始信息,B 和 C 为校验节点信息)以符号 $a_A$,$a_B$,$a_C$ 为地址分别写入单口 RAM A、B、C(记为 RAM. A/B/C,所有存储器均初始化为 0 值);然后分别读取节点 B/C 更新所需信息,具有相同地址的值直接相加,并将其和送入插值排序器,保留其中较大的 $n_m$ 个值及其符号(存储器地址)作为变量节点更新信息。当译码判决信号到达时,则直接将 RAM A、B、C 中具有相同地址的值直接相加,并将其和送入插值排序器,找出最大值对应符号作为译码输出。具体运算如下:

```
input:
for i = 1:n_m
    RAM. A[α_A] = A,RAM. B[α_B] = B,RAM. C[α_C] = C
end
sort:
for i = 1:16
    sort_in = RAM. A[i] + RAM. B[i] + RAM. C[i] → sort
    sort_in = RAM. A[i] + RAM. C[i] → sort
    sort_in = RAM. A[i] + RAM. B[i] → sort
end
initial:
for i = 1:16
    RAM. A[i] = 0,RAM. B[i] = 0,RAM. C[i] = 0
end
normalization:sort_out = sort_out - min(sort_out)
```

→ *sort* 表示将数据送入插值排序器,具体实现时,采用一个插值排序器以时分复用方式完成排序选择操作以节省资源。按照上述遍历方法,在插值排序运算结束后,需对 RAM 进行初始化操作,以保证下一列变量节点更新数据输入时,未出现符号对应的消息值为 0。在排序选出 $n_m$ 个最大消息值后,需对消息向量进行归一化处理以保证算法收敛性,设计中采取减去消息向量中最小(第 $n_m$ 个)元素的方法,并对消息值大于 16 的数置(8,4)量化所能表示的最大值(15.9375),仿真结果表明,上述处理对译码算法性能没有影响。图 10 给出了变量节点更新单元详细的硬件结构图。按照上述结构,对于度为 3 的变量节点更新完成一次迭代仅需 340 个时钟周期,对于度为 2 的变量节点更新完成一次迭代仅需 288 个时钟周期。

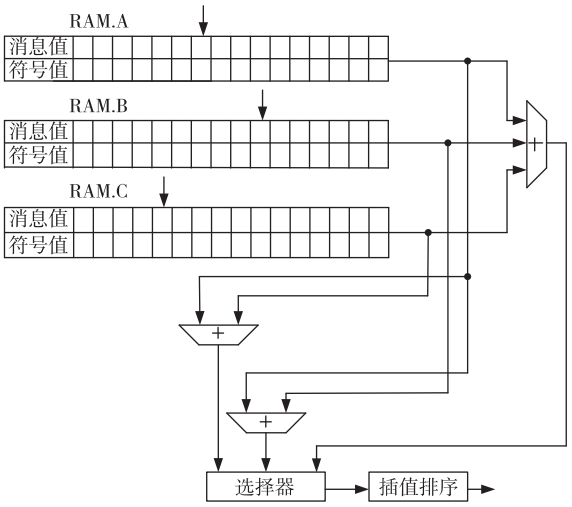


**图 10 提出的变量节点更新结构**

### 2.4 译码器信息调度

在标准信息调度算法中,变量/校验节点更新单元交替运行,变量节点更新时,校验节点更新单元处于闲置状态;校验节点更新时,变量节点更新单元处于闲置状态。这种调度算法具有地址生成简单、节点更新单元利用率较低等特点。图 11 描述了标准信息调度算法原理图,其中,$m$ 为校验节点更新单元校验节点处理器(CNP)完成一次迭代所需时钟周期,$n$ 为变量节点更新单元变量节点处理器(VNP)完成一次迭代所需时钟周期。

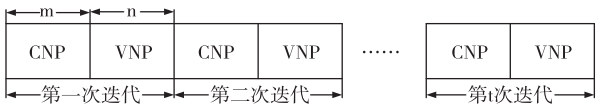


**图 11 标准信息调度算法原理图**

为提高变量/校验节点更新单元利用率,文献[9]提出了交叠的信息调度算法,其核心思想就是让校验/变量节点更新过程在时间上交叠,缩短译码

时间。即利用计算好的部分校验节点信息更新变量节点,而不是等待本次迭代校验节点更新结束之后再去更新变量节点。这样,校验/变量节点更新过程在时间上出现交集从而达到缩短译码时间的目的。该算法虽然一定程度提高了时钟增益,但其地址产生、读写控制较复杂。图 12 描述了交叠信息调度算法原理图,其中 $w$ 为校验/变量节点更新过程交叠时钟周期。

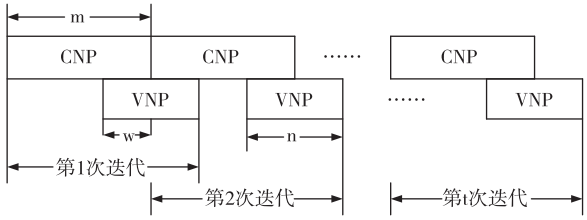


**图 12　交叠信息调度算法原理图**

从图 12 中可以看出,交叠信息调度算法相对标准信息调度算法的时钟增益为

$$\frac{clk_{交叠}}{clk_{标准}} = \frac{mt + n - w}{(m + n)t} \tag{8}$$

时钟增益主要受校验/变量节点更新过程交叠时钟周期影响。

针对上述问题,文献[10]提出了双进双出的信息调度算法,其核心思想是两帧数据同时译码。第一帧数据进行校验/变量节点更新时,第二帧数据进行变量/校验节点更新,以达到提高时钟增益,缩短译码时间的目的。两帧数据具有独立的变量/校验节点信息存储单元,校验/变量节点更新单元将更新结果交替写入两组存储单元。通常校验节点更新所需时钟周期 $m$ 不等于变量节点更新所需时钟周期 $n$,在标准双进双出的信息调度算法中,校验/变量节点更新分配的时钟周期必须取 $m$,$n$ 中的较大值,否则校验或变量节点更新单元将没有足够的时钟周期完成下一帧数据的更新。图 13 描述了此双进双出的信息调度算法原理图。

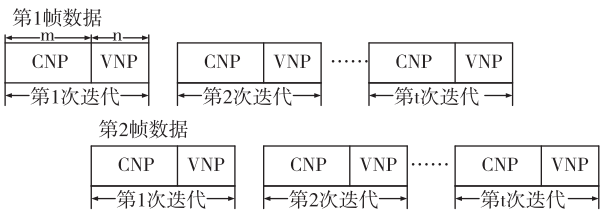


**图 13　双进双出信息调度算法原理图**

从图 13 中可以看出,交叠信息调度算法相对标准信息调度算法的时钟增益为

$$\frac{clk_{双进双出}}{clk_{标准}} = \frac{2mt + n}{(m + n)t} \times \frac{1}{2} = \frac{mt + n/2}{(m + n)t} \tag{9}$$

对比式(8)、(9),双进双出的信息调度算法与交叠的信息调度算法的时钟增益相近,但简化了地址生成和读写控制。

在多元 LDPC 译码器中,校验节点更新所需时钟周期远远大于变量节点更新,采用上述双进双出调度算法,相邻两次迭代 VNP 与 CNP 之间存在时间间隙,即 VNP 没有得到最大限度的利用,是一种资源浪费。本文采用 1 个 VNP 以复用的方式完成多列子矩阵的节点更新,使得 VNP 更新所需时钟周期接近 CNP 更新周期,缩小时间间隙的目的。本文设计中,一次迭代 CNP 需 $T_C = 758$ 个时钟周期完成一行子矩阵校验节点更新,VNP 需 $T_V = 340$ 个时钟周期完成一列子矩阵变量节点更新,$T_C \approx 2 \times T_V$。故本文设计中每 2 列子矩阵分配一个 VNP,相比一般结构,节省了 12 个 VNP。

## 3　设计结果

为了验证设计的多元 LDPC 译码器结构具有较高的译码吞吐量和较低的硬件实现复杂度,本文利用 Xilinx Virtex-4(XC4VLX200)平台对提出的结构进行了综合仿真,其中迭代次数为 5,消息向量长度 $n_m = 8$,并采用 12bit 量化(4bit 用于符号向量量化)。译码器吞吐量 *Throughput* 与码长 $N$、码率 $R$ 和时钟频率 $f$ 成正比关系,与迭代次数 *iter_num* 和单次迭代延时 *cycle* 成反比关系,其计算公式为

$$Throughput = \frac{N \times R \times f}{iter\_num \times cycle} \tag{10}$$

设计中分块矩阵维数为 5,16 元 LDPC 编码,码长 $24 \times 5 \times 4 = 480$bit,码率 3/4。在采取 5 次迭代情况下,由于采用双进双出信息调度算法,提出结构共需约 $(758 + 758) \times 5/2 + 758 = 4548$ 个周期完成一帧信号的译码。而采用前向后向算法设计的译码结构完成度为 3 的变量节点更新需 5 次单步运算(一次单步运算需约 41 个时钟周期完成,每列分块矩阵共 5 列),共需约 $(758 \times 3 + 5 \times 5 \times 41) \times 5 = 16495$ 个周期。即设计的译码器可达到 12Mbps 的吞吐量,约为一般译码结构的 3.6 倍,这主要是通过增加校验/变量节点更新并行度以及采用新的信息调度算法来获得的。表 1 列出了设计的译码器主要

性能参数。

**表 1 译码器主要性能参数**

| 性能指标 | 采用文中的译码结构设计的译码器 |
|---|---|
| 逻辑片 | 24546(27%) |
| 触发器 | 27161(15%) |
| 4 输入查找表 | 38528(21%) |
| 块 RAMs | 200(59%) |
| 最高频率/MHz | 152 |

吞吐量的提高是以硬件实现复杂度的增加为代价的,在 CNP 设计中,由于采用 3 路单步运算完成校验节点更新,其复杂度约为一般结构的 3 倍,整个校验节点更新相比一般结构多消耗了约 $12 \times 759 \approx 9000$ 个 slice(表 2 列出了本文设计校验节点更新单步运算单元资源消耗情况)。一般部分并行译码器结构共需 24 个 VNP 模块,采用本文结构共需 12 个 VNP 模块,则整个变量节点更新节省了约 $381 \times 24 - 257 \times 12 \approx 6000$ 个 slice(表 3 列出了两种变量节点更新复杂度对比),则整个译码器多消耗了约 3000 个 slice,即逻辑资源消耗增加了约 14%。

**表 2 译码器校验节点更新资源消耗**

| 性能指标 | 前向后向算法设计的 CNP 单步运算单元 |
|---|---|
| 逻辑片 | 759 |
| 触发器 | 741 |
| 4 输入查找表 | 1325 |

**表 3 译码器变量节点更新资源消耗**

| 性能指标 | 采用前向后向算法设计的 VNP | 本文设计的全并行 VNP |
|---|---|---|
| 逻辑片 | 381 | 257 |
| 触发器 | 383 | 314 |
| 4 输入查找表 | 569 | 374 |
| 单口 RAM | 2 | 3 |
| CAM | 1 | 0 |

## 4 结 论

本文设计了一种基于 EMS 算法的多元 LDPC 译码器。它根据前向后向算法规则,以 3 路单步运算单元完成校验节点更新;采用低复杂度全并行运算的变量节点信息更新单元,消除了变量节点更新的递归计算;采用新的双进双出信息调度算法,降低了变量节点更新复杂度。FPGA 仿真结果表明,提出的设计方法提高了多元 LDPC 译码器的吞吐量。

**参考文献**


[1] Davey M C, MacKay D. Low-density parity check codes over GF(q). *IEEE Communications Letters*, 1998, 2(6): 165-167

[2] Declercq D Fossorier M. Decoding algorithms for nonbinary LDPC codes over GF(q). *IEEE Transactions on Communications*, 2007, 55(4): 633-643

[3] Voicila A, Declercq D, Verdier F. Low complexity, low memory EMS algorithm for non-binary LDPC codes. In: Proceedings of International Conference on Communication, 2007, 671-676

[4] 刘飞, 黎海涛. 多元 LDPC 译码器设计. 信号处理, 2012, 28(3): 397-403

[5] 李博. 基于 EMS 算法的多元 LDPC 码译码器设计与 FPGA 实现. 西安: 西安电子科技大学通信工程学院, 2010

[6] 何光华, 白宝明, 李博. 采用 EMS 算法的多元 LDPC 译码器的 FPGA 实现. 西安电子科技大学学报, 2011, 38(5): 35-42

[7] Fang C. Effienct VLSI architecture for non-binary low density parity check decoding. America: CASE WESTERN RESERVE UNIVERSITY, 2011

[8] Voicila A, Declercq D, Verdier F. Architecture of a low-complexity non-binary LDPC decoder. In: Proceedings of International Conference on Consumer Electronics. 2008: 1-2

[9] Dai Y, Yan Z. Overlapped message passing for quasi-cyclic low-density parity check codes. *IEEE Transactions on Circuits and Systems I*, 2007, 51(6): 1106-1113

[10] 孙月. 多元 LDPC 码编译码器的设计与实现[博士学位论文]. 成都: 电子科技大学通信与信息工程学院, 2010

[11] Richardson T J, Urbanke R L. Efficient encoding of low-density parity-check codes. *IEEE Transactions on Information Theory*, 2001, 47(2): 638-656

[12] 吴晓丽. 多进制 LDPC 码的编译码算法及结构设计[博

士学位论文]. 西安:西安电子科技大学通信工程学院,2009

[13] Liu F,Li H T. Decoder design for nonbinary LDPC codes. In:Proceedings of IEEE International Conference on. In: Proceedings of IEEE International Conference on Wireless Communications, Networking and Mobile Computing, Wuhan,China,2011

# Design and implementation of non-binary LDPC decoder

Li Haitao[*] [**], Yang Leilei[*], Liu Fei[*], Yuan Haiying[*]

([*] College of Electronic Information and Control Engineering, Beijing University of Technology, Beijing 100124, China)

([**] Zhongxing Telecom Equipment Corporation, Beijing 100191)

**Abstract**

For the conventional extended min-sum (EMS) decoding algorithm for nonbinary quasicyclic LDPC(QC-LDPC) coder, it is necessary to perform quantities of recursive computation among the message vector, and only one single step operations is utilized to complete the check node update, which leads to larger decoder latency. In this paper, a novel non-binary LDPC decoder architecture is proposed to overcome this problem. Based on the rules of forward-backward algorithm, we utilize three single step operations to complete the check node update and optimize the check node update step operation. The hardware resource consumption for check node update increases, but the cycle required is reduced to 1/3 of the common decoder structure's. The variable node update unit with fully parallel computation is presented without forward-backward, which removes recursive computation among the message vector. Furthermore, double-input and double-output information scheduling algorithm is used to improve the throughput of the decoder and reduce the complexity of computation on variable node update further. Moreover, an FPGA implementation of a (480,360) nonbinary QC-LDPC decoder over GF(16) is designed to demonstrate the efficiency of the presented techniques. Simulation results show that the proposed scheme can triple throughput of the decoder at the cost of less hardware resource consumption.

**Keywords**: extended min-sum algorithm, non-binary low density parity check code, FPGA, throughput