논문 2005-42SD-1-9

Radix-4 비터비 디코더를 위한 효율적인 ACS 구조

(An Efficient ACS Architecture for radix-4 Viterbi Decoder)

김 덕 환*, 임 종 석**

(Deok-Hwan Kim and Chong-Suck Rim)

요 약

비터비 디코더는 통신 시스템에서 가장 핵심적인 부분 중의 하나로써 순방향 오류 정정을 위해 사용된다. 통신 속도의 고속화가 진행됨에 따라 고속에서 동작할 수 있는 통신 모듈의 개발이 점차 중요해지고 있다. 비터비 디코더는 궤환구조를 갖는 ACS 연산의 특성상 고속화가 매우 어렵다. 본 논문에서는 비터비 디코더의 고속화와 면적을 모두 고려한 효율적인 radix-4 ACS 구조를 제안하였다. 비터비 디코더의 ACS 연산을 재 정렬하여 면적을 절약하였고 경로 메트릭 메모리를 retiming하여 디코더의 속도를 개선하였다. 제안된 ACS 구조는 VHDL로 구현되었고 Xilinx의 ISE 6.2i에서 합성되었다. 실험을 통해서 제안된 구조의 AT product가 기존의 고속 radix-4 ACS 구조보다 11% 개선된 것을 확인할 수 있었다.

Abstract

The Viterbi decoder which is used for the forward error correction(FEC) is a crucial component for successful modern communication systems. As modern communication speed rapidly high, the development of high speed communication module is important. However, since the feedback loop in ACS operation, high speed of Viterbi decoder is very difficult. In this paper, we propose an area reduced, high speed ACS Architecture of Viterbi decoder based on the radix-4 architecture. The area is reduced by rearranging the ACS operations, and the speed is improved by retiming of path metric memory. The proposed ACS architecture of Viterbi decoder is implemented in VHDL and synthesized in Xilinx ISE 6.2i. The area-time product of the proposed architecture is improved by 11% compared to that of the previous high speed radix-4 ACS architecture.

Keywords: VLSI design, Viterbi decoding, high speed architecture.

I. 서 론

비터비 디코딩은 디지털 통신 시스템에서 순방향 에러 정정을 위해 가장 많이 사용되는 방법 중의 하나이다[10]. 통신 기술이 점차 발전됨에 따라 통신 속도의 고속화가 진행되고 이로 인해 고속으로 동작할 수 있는 비터비 디코더의 필요성이 점차 대두되고 있다. 그러나 비터비 디코더에서 얻을 수 있는 최대 throughput은 비선형 데이터 종속성의 순환구조를 갖는 ACS (add-compare-select) 연산에 의해서 제한을 받는다[5]. 따라서 고속으로 동작할 수 있는 비터비 디코더를 위해서는 효율적인 ACS 구조가 필요하다.

비터비 디코더의 throughput을 개선하는 방법은 크게 두 가지 계열로 나누어진다. 하나의 계열은 ACS 연산을 재구성 하거나 회로의 구조를 변형하여 회로의 임계 경로 지연 시간을 줄이는 방법이다. ACS 연산에서 비교 연산 시간을 줄이기 위한 병렬 비교 방법[2]이나 덧셈기에서 발생하는 지연 시간을 줄이기 위해 carry-save-adder를 사용하는 방법[5], 이중 상태 확장 방법[6], 비트 단위의 파이프라인 방법[7] 등이 이 계열에 속한다. 다른 하나는 비터비의 다중 입력 샘플에 대해서 동일한 ACS 연산을 병렬 수행하는 방법으로 nested-look-ahead 방법[8]이나 블록 디코딩 방법[9] 등이 이에 속한다. 그러나 이들 방법은 매우 큰 하드웨어 면적을 필요로 하기 때문에 매우 높은 throughput을 위해서만 사용될 수 있다.

본 논문에서는 radix-4 트렐리스 병렬 구조의 비터비

* 학생회원, ** 정회원, 서강대학교 컴퓨터학과
(Department of Computer Science, Sogang University)
접수일자: 2004년8월17일, 수정완료일: 2004년12월29일

디코더에 대해 면적과 속도의 개선을 모두 고려한 효율적인 ACS 구조를 제안하였다. 하드웨어 면적을 줄이기 위해 ACS 연산을 rearranging 하였고 이로 인해 늘어나는 임계 경로 지연 시간을 줄이기 위해 경로 메트릭 메모리를 retiming하는 방법을 이용하였다. 제안된 비터비 디코더는 VHDL로 구현되었고 Xilinx의 ISE 6.2i에서 합성되었다. 제안된 구조의 효율성을 정확히 비교하기 위해 구현된 회로의 게이트 카운트나 임계 경로 지연 시간 등을 측정하여 AT product를 기준으로 비교하였다. 실험을 통하여 제안된 ACS 구조가 기존의 ACS 구조보다 약 11% 적은 AT product를 가져 좀더 효율적인 구조임을 확인할 수 있었다.

본 논문에서 구현한 비터비 디코더는 구속장(K) 6, 부호화율 1/2에 맞게 구현되었고 추가적인 부호 이득(coding gain)을 얻기 위해 3-bit 연판정(soft decision) 방법을 사용하였다. 경로 메트릭의 overflow를 방지하기 위한 modulo arithmetic method[1]를 이용하기 위해 비터비 디코더에서의 모든 비교기를 ripple carry subtractor로 구현하였다.

본 논문의 구성은 다음과 같다. 먼저 Ⅱ절에서 비터비 디코더에 대한 기본적인 개념을 정리하고 기존에 알려진 radix-4 트렐리스 병렬 구조의 ACS 구조에 대해 살펴본다. 다음 Ⅲ절에서는 본 논문에서 제안한 ACS 구조에 대해서 설명하고 Ⅳ절에서 제안된 구조와 기존의 구조의 비교 결과를 살펴본다. 마지막으로 Ⅴ장에서 본 논문의 결론을 내린다.

Ⅱ. 기본정리

1. 비터비 알고리즘.

비터비 알고리즘은 Most-likely-State-Sequence를 찾는 알고리즘으로 1976년에 처음 제안되었다. 비터비 알고리즘에 의한 복호과정은 트렐리스도를 이용하여 간결하게 설명될 수 있다. 그림 1에 4개의 상태를 가진 트렐리스도를 보인다. 그림 1에서 보듯이 시간에 따라 각 상태는 다른 상태로 천이(transition)가 일어날 수 있다. 각 천이는 연판정 (soft decision) 또는 경판정 (hard decision)의 적용에 따라 유클리드거리 (euclidean distance)나 해밍거리 (hamming distance)에 의해 계산된 가지 메트릭 (branch metric)을 갖는다. 임의의 시간에서 각 상태까지 오기 위해 결정된 경로를 생존경로(survivor path)라고 하고 이들 생존경로에 따라 누적된 가지 메트릭을 경로 메트릭(path metric)또는 상태 메트릭(state metric)이라고 한다.

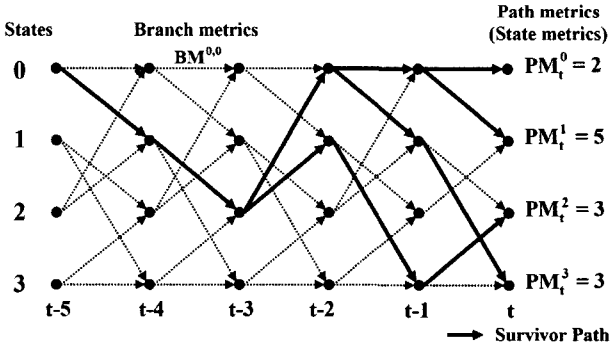

그림 1. Trellis diagram
Fig. 1. Trellis diagram.

그림 1에서 가지 메트릭 $BM^{i,j}$는 상태 i에서 상태 j로의 천이에 대한 가지 메트릭을 의미하고 PM_t^s는 시간 t에 상태 s에서의 경로 메트릭을 의미한다. 비터비 알고리즘은 주어진 트렐리스도에서 각 상태에 대해 ACS (Add-Compare-Select)연산이라고 하는 식 (1)을 수행하여 각 상태의 경로 메트릭을 갱신하고 생존 경로를 결정한다. 각 상태에서 생존 경로에 대한 결정은 이전 상태의 경로 메트릭과 가지 메트릭을 더한 값 중 가장 작은 값을 갖는 경로를 선택함으로써 이루어지게 되는데 이러한 선택을 ACS decision이라 하고 이를 임의의 비트로 표현한 것을 ACS decision bits라 한다.

$$PM_t^k = min\{PM_{t-1}^i + BM^{i,k},\ PM_{t-1}^j + BM^{j,k}\} \quad (i,j,k \in \{0,1,2,3\}) \tag{1}$$

비터비 알고리즘의 복호화는 각 상태에 대한 생존 경로 중 송신 데이터와 가장 유사한 생존 경로를 찾아내고 이를 역으로 추적함으로서 이루어진다.

비터비 디코더는 크게 아래와 같은 3개의 부분으로 이루어지는데, 각 입력 심볼에 대해 가지 메트릭을 계산하는 부분을 BMU(Branch Metric Unit), 각 상태에 대해 경로 메트릭의 갱신과 생존 경로에 대한 결정을 수행하는 부분을 ACSU(ACS Unit), ACSU로부터 얻어진 ACS decisions을 이용하여 생존 경로를 역으로 추적하는 부분을 SMU(Survivor Metric Unit)라고 한다.

2. Radix-4 트렐리스 비터비 디코더의 ACS 구조.

일반적으로 VLSI의 구현에서 throughput과 칩 면적은 중요한 고려대상이다. 특히 비터비 디코더의 구현에서 비선형 데이터 종속성의 특성을 갖는 ACS 순환구조는 throughput을 제한하는 중요한 요인이었다. 이들 ACS의 순환구조를 고려하여 throughput을 높이는 가

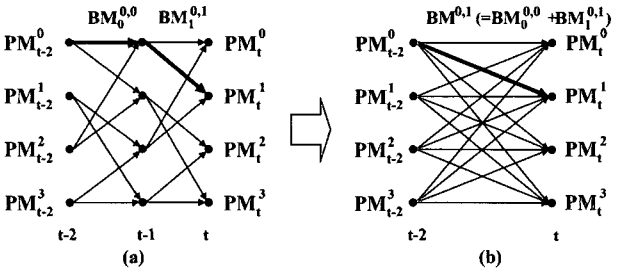

그림 2. (a) Radix-2 trellis, (b) Radix-4 trellis
Fig. 2. (a) Radix-2 trellis, (b) Radix-4 trellis.

장 대표적인 방법으로는 M-step의 트렐리스를 1-step의 트렐리스로 변환하여 ACS 연산을 수행하는 것이다. 그러나 M이 높은 경우 개선된 throughput에 비해 회로의 복잡도에 의한 칩 면적의 증가량이 매우 커지게 된다. 따라서 본 논문에서는 가장 많이 쓰이는 2-step 트렐리스의 radix-4 병렬 구조를 구현 대상으로 하였다.

그림 2에서 보듯이 2-step의 radix-2 트렐리스도는 1-step의 radix-4 트렐리스도로 변환될 수 있다. 이때 radix-4 트렐리스도에서의 각 상태는 4개의 모든 상태로 천이가 일어날 수 있고 각 천이에 대한 가지 메트릭 $BM^{i,j}$은 radix-2 트렐리스도에서의 가지 메트릭들 $(BM_0^{i,l}, BM_1^{l,j})$의 합으로 이루어지게 된다.

일반적으로 비터비 디코더의 VLSI구현에서 ACSU는 경로 메트릭 메모리의 access 횟수를 줄이기 위해 butterfly 구조로 구현된다[3]. 그림 2(b)에서 보듯이 경로 메트릭 PM_t^0을 갱신하기 위해서는 이전 경로 메트릭들$(PM_{t-2}^0, PM_{t-2}^1, PM_{t-2}^2, PM_{t-2}^3)$과 가지 메트릭들$(BM^{0,0}, BM^{1,0}, BM^{2,0}, BM^{3,0})$을 더한 값 중 가장 최소의 값을 찾는 연산을 수행해야 한다. 이 연산은 하나의 ACS-block으로 구현될 수 있고 4개의 각 경로 메트릭 PM_t^0, PM_t^1, PM_t^2, PM_t^3를 모두 갱신하기 위해서는 4개의 ACS-block이 필요하게 된다. 이렇게 4개의 ACS-block으로 이루어진 구조를 radix-4 ACSU butterfly 구조라고 한다. 그림 3에 일반화된 radix-4 butterfly 구조를 보인다. 그림 3에서 S는 트렐리스도에 있는 상태의 총 개수를 의미 한다.

일반적으로 N개의 값 중 가장 작은 값 또는 가장 큰 값을 선택하는 문제는 $log_2 N$ 시간을 갖는 비교 방법으로 해결될 수 있다. 이 방법을 이용한 ACS 구조를 conventional radix-4 architecture(CR4A)라고 하고 그림 4에 CR4A의 ACS-block 구조를 보인다. 그림 4에서 보듯이 CR4A에서는 이전 경로 메트릭과 가지 메트릭이 더해지고 이들 메트릭중 가장 작은 메트릭을 선택하

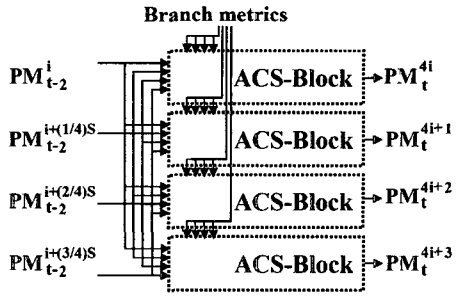

그림 3. Butterfly structure of radix-4 ACSU
Fig. 3. Butterfly structure of radix-4 ACSU.

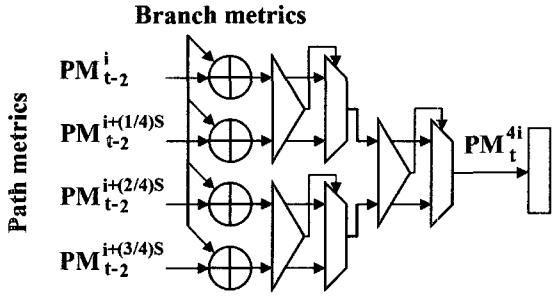

그림 4. ACS-block structure of CR4A
Fig. 4. ACS-block structure of CR4A.

기 위해 비교기와 multiplexer가 트리형태로 구성된다.

radix-4에서 ACS 연산을 수행하기 위한 다른 방법이 [2]에서 제안되었다. 그림 4에서 보듯이 CR4A에서는 최소 메트릭을 선택하기 위한 비교 연산이 2-step으로 이루어진다. 그러나 [2]에서 제안된 방법은 단지 1-step의 비교 연산만으로 최소 메트릭을 선택한다. 이는 각 경로 메트릭들과 가지 메트릭들에 대해서 아래 식 (2)와 같은 6개의 비교 연산을 수행함으로서 이루어질 수 있다. 아래 식 (2)는 경로 메트릭 PM_t^{4i}를 갱신하기 위한 ACS-block에서 수행되는 6개의 비교 연산이다. 이런 비교 방법을 이용한 ACS 구조를 fast radix-4 architecture(FR4A)라고 하고 그림 5에 ACS-block 구조를 보인다. 그림 5에서 보듯이 메트릭의 비교가 1-step으로 이루어지기 때문에 CR4A보다 빠른 속도로 동작할 수 있고 따라서 비터비 디코더의 throughput은 높아지게 된다.

$$
\begin{aligned}
&PM_{t-2}^{i} + BM^{i,4i} > PM_{t-2}^{i+(2/4)S} + BM^{i+(2/4)S,4i} \\
&PM_{t-2}^{i} + BM^{i,4i} > PM_{t-2}^{i+(1/4)S} + BM^{i+(1/4)S,4i} \\
&PM_{t-2}^{i} + BM^{i,4i} > PM_{t-2}^{i+(3/4)S} + BM^{i+(3/4)S,4i} \\
&PM_{t-2}^{i+(2/4)S} + BM^{i+(2/4)S,4i} > PM_{t-2}^{i+(1/4)S} + BM^{i+(1/4)S,4i} \\
&PM_{t-2}^{i+(2/4)S} + BM^{i+(2/4)S,4i} > PM_{t-2}^{i+(3/4)S} + BM^{i+(3/4)S,4i} \\
&PM_{t-2}^{i+(1/4)S} + BM^{i+(1/4)S,4i} > PM_{t-2}^{i+(3/4)S} + BM^{i+(3/4)S,4i}
\end{aligned}
\qquad (2)
$$

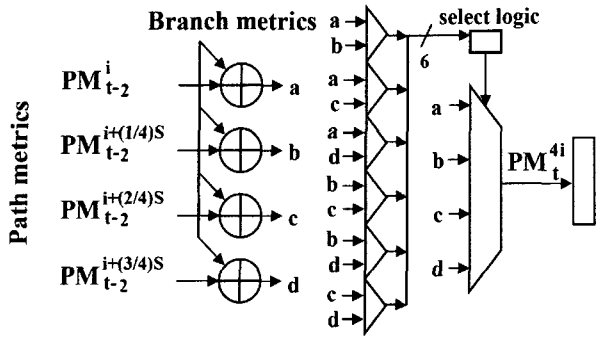

그림 5. ACS-block structure of FR4A
Fig. 5. ACS-block structure of FR4A.

앞에서 설명된 두 ACS 구조에 대한 하드웨어 사이즈의 비교는 간단히 각 butterfly 구조에서 사용되는 연산기의 개수 비교를 통하여 비교될 수 있다. CR4A의 butterfly 구조에서는 16개의 덧셈기와 12개의 비교기, 12개의 2:1 multiplexer가 사용되고 FR4A의 butterfly 구조에서는 16개의 덧셈기와 24개의 비교기, 4개의 4:1 multiplexer가 사용된다. 따라서 FR4A는 1-step의 비교연산으로 인하여 CR4A보다 속도가 빠른 반면 이로 인해 비교기의 개수가 증가하여 더 큰 하드웨어 면적을 필요로 한다.

3. ACS 연산의 rearranging.

radix-2 ACSU butterfly 구조는 두개의 ACS-block으로 구성되고, 각 두 ACS-block에서 수행되는 비교연산은 식 (3)과 같다. 식 (3)에서 보듯이 이 butterfly 구조에서는 4개의 덧셈기와 2개의 비교기가 사용된다.

$$\begin{aligned} PM^{i}_{t-1} + BM_0^{i,2i} &> PM^{i+S/2}_{t-1} + BM_0^{i+S/2,2i} \\ PM_{t-1} + BM_0^{i,2i+1} &> PM^{i+S/2}_{t-1} + BM_0^{i+S/2,2i+1} \end{aligned} \quad (3)$$

$$\begin{aligned} PM^{i}_{t-1} - PM^{i+S/2}_{t-1} &> BM_0^{i+S/2,2i} - BM_0^{i,2i} \\ PM^{i}_{t-1} - PM^{i+S/2}_{t-1} &> BM_0^{i+S/2,2i+1} - BM_0^{i,2i+1} \end{aligned} \quad (4)$$

식 (3)은 식 (4)로 rearranging될 수 있다[4]. 식 (4)에서 오른쪽의 가지 메트릭의 차이 값들은 BMU에서 미리 계산되어지고 ACSU의 모든 ACS-block에서 공유될 수 있다. 왼쪽의 경로 메트릭 차이 값들은 ACSU에서 계산되고 식 (4)를 수행하는 ACS-block들 사이에서 공유될 수 있다. 식 (3)와 식 (4)는 본질적으로 같은 비교연산이므로 같은 비교 결과를 갖게 된다. 그러나 식 (3)에서와는 달리 식 (4)에서는 얻어진 ACS decision에 의해 새로운 경로 메트릭이 선택되는 것이 아니라 경로 메트릭을 갱신하기 위한 가지 메트릭과 이전 시간의 경로 메트릭이 선택되어진다. 따라서 이들을 더하기 위한 덧셈기가 각 ACS-block마다 추가적으로 필요하게 된다[4]. 가지 메트릭 차이 값을 계산하기 위한 cost를 제외하면 식 (4)의 butterfly 구조에서는 경로 메트릭 차이 값을 계산하기 위한 subtractor 1개와 선택된 가지 메트릭과 이전 시간의 경로 메트릭을 더하기 위한 덧셈기 2개가 사용되어 식 (3)의 구조보다 덧셈기를 하나 절약할 수 있다.

Ⅲ. 면적 절약형 Radix-4 ACSU 구현

1. 면적 절약을 위한 rearranging 방법의 적용.

본 논문에서는 ACSU에서의 면적 절약을 위해 [4]에서 제안된 rearranging 방법을 FR4A의 ACS 구조에 적용시켰다. FR4A에서 수행되는 비교 연산인 식 (2)는 rearranging되어 식 (5)로 변환된다. 식 (5)를 비교 연산으로 수행하는 ACS 구조를 rearranged radix-4 architecture(RR4A)라고 한다.

$$\begin{aligned} PM^{i}_{t-2} - PM^{i+(2/4)S}_{t-2} &> BM^{i+(2/4)S,4i} - BM^{i,4i} \\ PM^{i}_{t-2} - PM^{i+(1/4)S}_{t-2} &> BM^{i+(1/4)S,4i} - BM^{i,4i} \\ PM^{i}_{t-2} - PM^{i+(3/4)S}_{t-2} &> BM^{i+(3/4)S,4i} - BM^{i,4i} \\ PM^{i+(2/4)S}_{t-2} - PM^{i+(1/4)S}_{t-2} &> BM^{i+(1/4)S,4i} - BM^{i+(2/4)S,4i} \\ PM^{i+(2/4)S}_{t-2} - PM^{i+(3/4)S}_{t-2} &> BM^{i+(3/4)S,4i} - BM^{i+(2/4)S,4i} \\ PM^{i+(1/4)S}_{t-2} - PM^{i+(3/4)S}_{t-2} &> BM^{i+(3/4)S,4i} - BM^{i+(1/4)S,4i} \end{aligned} \quad (5)$$

Ⅱ.3에서 설명되었듯이 식 (5)에서 왼쪽의 경로 메트릭 차이 값들은 경로 메트릭 PM_t^{4i}, PM_t^{4i+1}, PM_t^{4i+2}, PM_t^{4i+3}을 갱신하기 위한 ACS-block들 사이에 모두 공유될 수 있고 오른쪽의 가지 메트릭 차이 값은 모든 ACS-block에서 공유될 수 있다. 식 (2)와 식 (5)는 본질적으로 같은 비교 연산이기 때문에 각 ACS-block에서의 ACS decision은 FR4A에서와 같은 방법으로 얻을 수 있다. 그러나 RR4A에서는 경로 메트릭을 갱신하는 과정이 FR4A와는 다르다. 그림 5에서 보듯이 FR4A에서는 먼저 이전 경로 메트릭과 가지 메트릭이 더해진다. 그리고 이들 더해진 메트릭들은 갱신될 경로 메트릭의 후보 메트릭이 되어 이들 중 하나가 비교 연산 후 얻어진 ACS decision에 의해 생존 경로 메트릭으로 선택되어진다. 그러나 RR4A에서는 FR4A에서와는 다르게 4개의 후보 메트릭들이 미리 계산되지 않는다. 그림 7에서 보듯이 비교 연산으로 얻어진 ACS decision에 의해 가지 메트릭과 이전 시간의 경로 메트릭이 선택되어지고, 선택된 메트릭들은 경로 메트릭 메모리에 저장되지 전에 더해지게 된다. 따라서 Ⅱ.3에서와 마찬가지

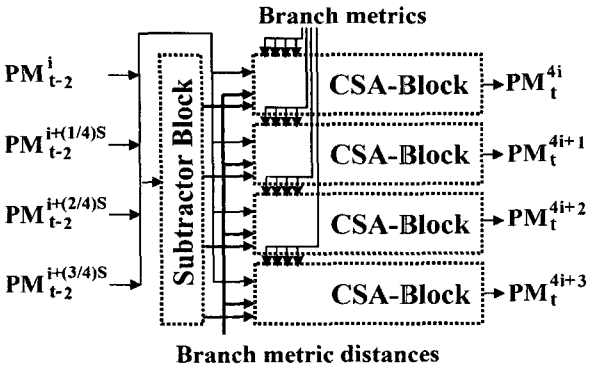

그림 6. Butterfly structure of RR4A
Fig. 6. Butterfly structure of RR4A.

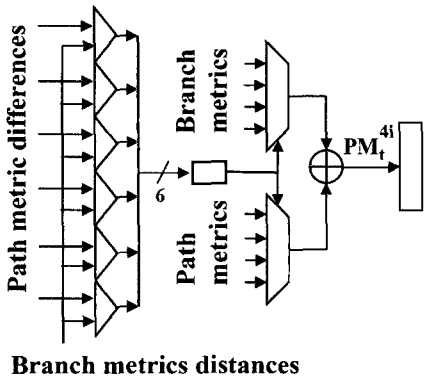

그림 7. CSA-block of RR4A
Fig 7. CSA-block of RR4A.

로 경로 메트릭을 갱신하기 위해 각 ACS-block마다 덧셈기가 하나씩 필요하게 된다. 이렇게 연산 순서가 다른 RR4A에서의 연산을 FR4A에서의 연산과 구분하기 위해 CSA (Compare-Select-Add) 연산이라고 하고 아래 그림 6과 7에 RR4A의 butterfly구조와 CSA-block 구조를 보인다. 그림 6에서 subtractor-block은 경로 메트릭의 차이값을 계산하기 위한 6개의 subtractor로 이루어져 있고 각 subtractor의 결과는 BMU에서 미리 계산된 가지 메트릭 차이 값과 함께 4개의 CSA-block으로 들어가게 된다. 그림 7에서 CSA-block은 가지 메트릭 차이 값과 경로 메트릭 차이 값을 비교하기위한 6개의 비교기와 6개의 비교 결과로부터 ACS decision을 얻어내는 select logic, 생존 경로 메트릭을 얻기 위한 덧셈기로 이루어진다. CR4A나 FR4A의 butterfly 구조에서 16개의 덧셈기가 사용되었던 것 과는 달리 RR4A의 butterfly 구조에서는 6개의 subtractor와 4개의 덧셈기가 사용되어 덧셈기의 개수가 절약되었다.

FR4A의 butterfly 구조에서는 비교 연산을 위해 24개의 비교기가 사용되었다. 그러나 RR4A에서는 사용되는 비교기의 개수를 20개로 줄일 수 있다. Radix-4에서 가지 메트릭 $BM^{i,4i}$, $BM^{i,4i+1}$, $BM^{i,4i+2}$, $BM^{i,4i+3}$은 아래 식 (6)과 같이 radix-2에서의 가지 메트릭(BM_0, BM_1)들의 합으로 이루어진다.

$$\begin{aligned} BM^{i,4i} &= BM_0^{i,2i} + BM_1^{2i,4i} \\ BM^{i,4i+1} &= BM_0^{i,2i} + BM_1^{2i,4i+1} \\ BM^{i+(2/4)S,4i} &= BM_0^{i+(2/4)S,2i} + BM_1^{2i,4i} \\ BM^{i+(2/4)S,4i+1} &= BM_0^{i+(2/4)S,2i} + BM_1^{2i,4i+1} \end{aligned} \quad (6)$$

위의 식 (6)에 의해서 두 가지 메트릭 차이값 $BM^{i+(2/4)S,4i} - BM^{i,4i}$와 $BM^{i+(2/4)S,4i+1} - BM^{i,4i+1}$은 모두 같은 값 $BM_0^{i+(2/4)S,2i} - BM_1^{i,2i}$을 갖는다. 경로 메트릭 PM_t^{4i}와 PM_t^{4i+1}를 갱신하기 위한 두 ACS-block에서 총 12개의 비교 연산이 수행되는데 그 중 아래의 두 비교 연산은 가지 메트릭 차이 값이 모두 같은 값을 가지기 때문에 같은 결과를 갖게 된다.

$$\begin{aligned} PM_{t-2}^{i} - PM_{t-2}^{i+(2/4)S} &> BM^{i+(2/4)S,4i} - BM^{i,4i} \\ PM_{t-2}^{i} - PM_{t-2}^{i+(2/4)S} &> BM^{i+(2/4)S,4i+1} - BM^{i,4i+1} \end{aligned}$$

따라서 두 비교 연산 중 한 연산은 수행될 필요가 없다. 이렇게 같은 가지 메트릭 차이 값으로 인해 중복되는 연산은 24개중 총 8개가 존재하고 이중 4개의 비교 연산은 수행될 필요가 없다. 따라서 RR4A의 butterfly 구조에서는 이런 성질을 이용하여 24개의 비교기가 아니라 총 20개의 비교기가 사용되고 이로 인해 ACSU에서 면적이 감소하게 된다.

RR4A에서는 각 비교기의 면적이 감소될 수 있다. 식 (6)에서 경로 메트릭 차이 값은 8bits이고 가지 메트릭 차이 값은 6bits이다. 그러나 식 (5)의 경우처럼 두 가지 메트릭 $BM^{i,4i}$와 $BM^{i+(2/4)S,4i}$이 모두 radix-2에서의 가지 메트릭 $BM_1^{2i,4i}$을 공통으로 가지고 있고, 이 메트릭이 가지 메트릭 차이 값 $BM^{i+(2/4)S,4i} - BM^{i,4i}$을 계산하는 과정에서 없어지게 되는 경우에는 가지 메트릭 차이 값이 6bits이 아니라 5bits이 된다. 따라서 FR4A의 butterfly 구조에서는 24개의 8bits:8bits 비교기가 사용되는 반면, RR4A에서는 16개의 8bits:6bits 비교기와 4개의 8bits:5bits 비교기가 사용된다. 이렇게 bit-width가 작아진 메트릭을 비교하는 비교기의 면적을 줄이기 위해 [4]에서 제안된 비교 방법을 이용하였다. [4]에서는 서로 다른 bit-width를 갖는 두 데이터의 비교시, 비교기에서 발생하는 전력 소모를 최소화하기 위한 데이터 분할 비교 방법과 2-스테이지 파이프라인 방법을 제안하였다. 본 논문에서는 데이터 분할 비교 방법만을

이용하여 각 8bits:6bits 비교기와 8bits:5bits 비교기의 면적을 절약하였다. 따라서 RR4A에서는 가지 메트릭 차이 값의 줄어든 bit-width로 인해 각 비교기에서의 추가적인 면적 감소가 이루어진다.

2. 속도 개선을 위한 retiming 방법의 적용.

앞에서 설명되었듯이 ACS 연산을 rearranging하여 ACSU에서 면적의 감소를 얻을 수 있었다. 그러나 이로 인해 ACSU에서 임계경로 지연시간이 늘어나게 되어 비터비 디코더의 throughput은 줄어들게 된다. 본 논문에서는 경로 메트릭 메모리를 retiming하여 ACSU에서의 임계경로 지연시간을 개선하였다.

일반적으로 2개의 n-bit ripple carry adder가 직렬로 연결되어 두 adder에서 carry-propagation이 병렬로 진행되는 경우 그 지연시간은 n+1 bit ripple carry adder의 지연시간과 같다[2]. 본 논문에서 사용된 각 비교기는 ripple carry subtractor로 구성되어있기 때문에 n-bit ripple carry adder와 비교기가 직렬로 연결되어 있는 경우 같은 지연시간 추정 방법이 적용될 수 있다. 이와 같은 추정 방법을 이용하여 각 ACS 구조 CR4A, FR4A, RR4A의 추정된 임계경로 지연시간을 그림 8에 보인다. 간단한 비교를 위해 ripple carry adder의 전가산기 지연시간과 ripple carry subtractor의 carry logic 지연시간이 같다고 가정하였다.

그림 8에서 보듯이 RR4A는 FR4A보다 8bits ripple carry adder의 지연시간을 더 갖는 반면 CR4A와 거의 비슷한 지연시간을 갖는다. RR4A에서 지연시간이 증가한 이유는 앞에서 언급되었듯이 비교 연산과 선택 연산 이후 최종 경로 메트릭을 얻기 위한 덧셈연산이 추가적으로 필요하기 때문이다. 그러나 앞에서 언급된 직렬연결의 덧셈기에 관한 지연시간의 특성을 이용하면 RR4A에서 늘어난 임계경로 지연시간을 줄일 수 있다. 그림 7에서 보듯이 RR4A에서 경로 메트릭 메모리는 최종 경로 메트릭을 계산하는 덧셈기의 뒤에 놓여 있다. 그러나 만약 그림 9(a)와 같이 경로 메트릭 메모리와 ACS decision 메모리를 덧셈기 앞으로 이동시키면 최종 경로 메트릭을 위한 덧셈기와 subtractor-block의 subtractor, CSAblock의 비교기가 모두 직렬로 연결된다. 따라서 이들 세 부분의 전체 지연시간은 10bits ripple carry adder의 지연시간과 같게 되고 그림 9(b)에서 보듯이 전체 임계경로 지연시간은 줄어들게 된다. 이렇게 경로 메트릭 메모리를 retiming한 ACS 구조를 retiming rearranged radix-4 architecture (RRR4A)라

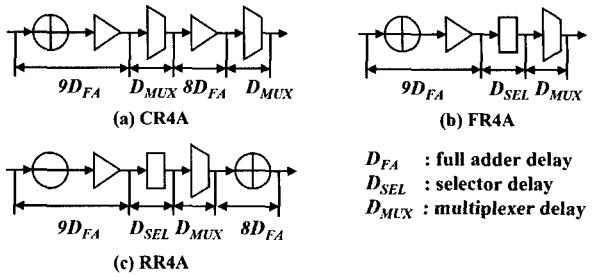

그림 8. Comparison of delay of each architectures
Fig. 8. Comparison of delay of each architectures.

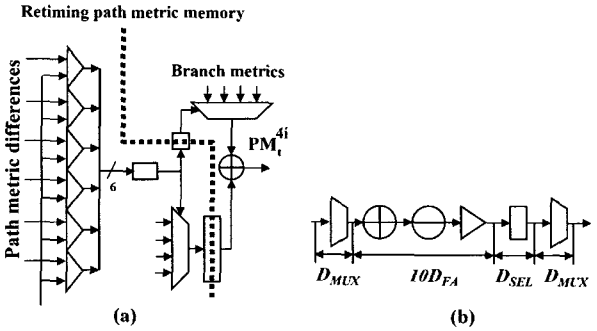

그림 9. (a) Retiming of RR4A, (b) delay of RRR4A
Fig 9. (a) Retiming of RR4A, (b) delay of RRR4A.

표 1. Comparison of critical path delay
Table 1. Comparison of critical path delay.

Architecture	Delay
CR4A	$17D_{FA}+2D_{MUX}$
FR4A	$9D_{FA}+D_{SEL}+D_{MUX}$
RR4A	$17D_{FA}+D_{SEL}+D_{MUX}$
RRR4A	$10D_{FA}+D_{SEL}+2D_{MUX}$

고 하고 각 네 개의 서로 다른 구조에 대한 임계경로 지연 시간의 비교를 표 1에 보인다. 표 1에서 보듯이 덧셈기와 subtractor, 비교기의 직렬연결로 인해 RRR4A는 FR4A보다는 크지만 CR4A보다는 작은 지연시간을 갖는다.

제안된 retiming으로 인해 지연 시간은 개선되었지만 이로 인해 추가적인 게이트의 증가가 일어난다. 그림 9(a)에서 보듯이 최종 경로 메트릭의 갱신을 위한 덧셈연산이 1 클럭 뒤에 일어나기 때문에 덧셈기에 공급되는 가지 메트릭 역시 1 클럭 지연되어야 한다. radix-4에서 가지 메트릭들의 조합은 총 16가지가 있고 각 가지 메트릭이 5bits이므로 이 지연을 위해 총 80bits의 register가 추가되어야 한다.

3. SDU에서의 면적 감소를 위한 selection logic.

SDU(State Decode Unit)는 비터비 디코딩 과정에서 추가적인 BER 이득을 얻기 위해 사용된다. 그림 10에

서 보듯이 일반적으로 SDU는 ACSU 뒤에 위치하여 ACSU에 의해 갱신되는 경로 메트릭 메모리로부터 최소 경로 메트릭을 가지는 상태(state)를 찾아 이를 SMU에 전달하는 Unit이다. 전달된 상태는 traceback 연산의 시작 위치로 사용되고, 이로부터 추가적인 BER 이득이 얻어진다.

일반적으로 SDU는 고속의 동작을 위해 병렬 구조의 비교기 트리와 파이프라인을 이용하여 구현되는데 병렬 구조의 비교기 트리는 상태의 개수가 S일 때 S-1개의 비교기로 구성되고 고속의 동작을 위해 $log_2 S$개의 파이프라인 스테이지로 나누어진다. 그림 10에서 보듯이 SDU가 P개의 파이프라인 스테이지로 나누어지는 경우 SMU로 전달되는 ACS decision bits도 P 클럭만큼 지연되어야 한다. 이는 SDU의 결과와 ACS decisions사이의 동기를 맞추기 위해서 인데 이로 인해 P만큼의 지연 register가 추가로 사용되어야 한다. 따라서 SDU에서의 파이프라인 수가 증가하면 SDU에서 뿐만이 아니라 ACS decision bits을 지연시키기 부분에서도 면적이 증가하게 된다. RRR4A에서는 ACSU에 약간의 logic을 추가하여 SDU에서의 비교기 트리와 파이프라인 스테이지 수를 감소시킬 수 있다. 따라서 이로 인해 SDU에서와 지연 register에서 면적을 절약할 수 있다.

그림 11에서 보듯이 각 덧셈기로부터 현재 경로 메트릭들 a, b, c, d가 계산되고 이들은 subtractor-block으로 전달된다. subtractor-block에서는 경로 메트릭 차이 값을 계산하기 위한 6개의 subtractor가 있는데 이들 각 subtractor결과의 MSBs를 이용하면 a, b, c, d 중 최소 메트릭을 찾을 수 있다. 이렇게 subtractor의 MSBs를 이용하여 최소 메트릭을 찾는 logic을 minimum selection logic이라고 하고 이를 포함한 ACS 구조를 그림 11에 보인다. minimum selection logic을 포함한 RRR4A의 ACS 구조에서는 SDU이 단지 (S/4)-1개의 비교기와 $log_2 S/4$의 파이프라인 스테이지로 구성될 수 있고 이로 인해 SDU에서와 ACS decision bits의 지연을 위한 지연 register에서 면적이 감소될 수 있다. 그러나 minimum selection logic에 의해 선택된 최소 메트릭이 ACS decisions보다 1 클럭 이전 시간의 메트릭이기 때문에 ACS decision bits는 1 클럭 더 지연되어야 하고 이를 위해 64-bits의 지연 register가 추가로 필요하게 된다.

본 논문에서 설계한 32-state ACS 구조의 경우는 8개의 minimum selection logic을 추가하여 SDU에서 24개의 비교기와 1-스테이지의 파이프라인, ACS deci-sions 지연을 위한 64-bits의 지연 register를 줄일 수 있었다.

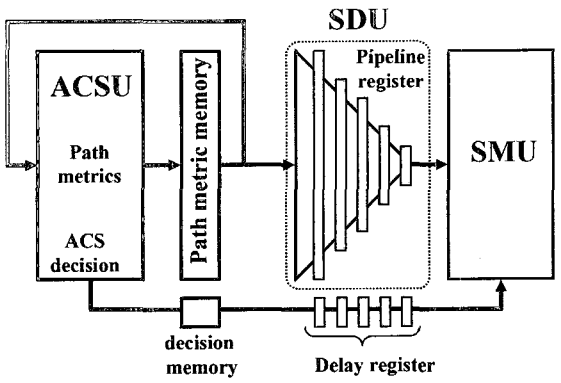

그림 10. The structure of ACSU, SDU, SMU
Fig. 10. The structure of ACSU, SDU, SMU.

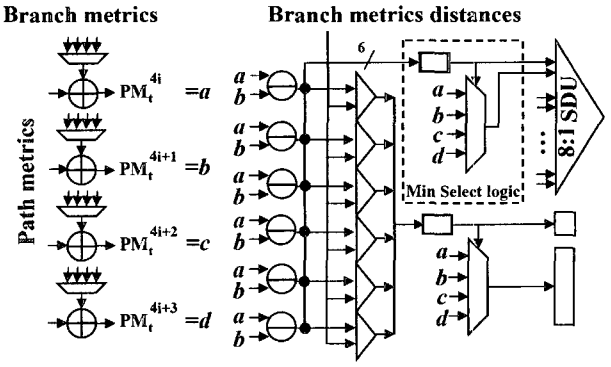

그림 11. RRR4A with reduced SDU
Fig. 11. RRR4A with reduced SDU.

Ⅳ. 실험 결과

각 ACS 구조 CR4A, FR4A, RRR4A가 VHDL로 구현되었고, Xilinx ISE 6.2i를 이용하여 합성되었다[11]. 각 구조의 정확한 비교를 위해 구현된 비터비 디코더 중 SMU을 제외한 나머지 부분만을 비교 하였다. 표 2에 구현된 각 구조의 게이트 카운트와 임계경로 지연시간(ns)을 보이고, 또한 FR4A와 RRR4A에 대해서 CR4A를 기준으로 한 게이트 카운트의 증가량과 임계경로 지연시간의 감소량을 보인다.

표 2에서 보듯이 FR4A의 게이트 카운트는 CR4A와 비교 약 32.94%가 증가되었고 임계경로의 지연시간은 약 23.19% 감소되었다. 반면 본 논문에서 제안된 구조인 RRR4A는 CR4A보다 약 20.16%정도 게이트가 증가한 반면 지연시간은 약 16.74% 감소되었다. FR4A와 비교, RRR4A는 CR4A 대비 임계경로 지연시간의 감소량이 6.45% 큰 반면 게이트 카운트 증가량은 12.78%정도

표 2. Comparison results of gate counts, delays
Table 2. Comparison results of gate counts, delays.

Architecture	CR4A	FR4A	RRR4A
Gate Counts(GC)	36,713	48,809	44,117
% increase of GC		32.94	20.16
Critical path delay(CD)	17.634	13.543	14.681
% reduction of CD		23.19	16.74

표 3. AT products of each architectures
Table 3. AT products of each architectures.

Architecture	Area (# of slices)	Time (cycle time)	AT product
CR4A	3494	17.634	1
FR4A	4966	13.543	1.09
RRR4A	4125	14.681	0.98

작은 것을 볼 수 있다.

좀더 정확한 비교를 위해 AT product를 이용하여 각 세 구조를 비교하였다. AT product는 하드웨어 효율성을 나타내는 방법 중의 하나로 하드웨어의 면적(A:Area)과 데이터 처리 주기(T:time)를 이용하여 비교하는 방법이다. 구현된 각 구조가 Xilinx의 ISE에서 합성되었기 때문에 면적(A)의 기준으로 각 구조의 구현에 사용된 FPGA의 slices개수를 사용하였고, 시간(T)의 기준으로 클럭의 주기 시간(=critical path delay)을 사용하였다. 일반적으로 slices는 FPGA에서 가장 기본적인 하드웨어 구성단위이다[11]. 세 구조에 대해 AT product를 이용하여 비교한 결과를 표 3에 보인다.

표 3에서 각 구조의 AT product는 CR4A를 기준으로 정규화 되었다. 표 3에서 보듯이 FR4A의 AT product가 CR4A보다 9% 높은 반면 RRR4A는 CR4A보다 2%정도 작아 세 구조 중에 가장 작은 AT product를 갖는 것을 볼 수 있다.

V. 결 론

본 논문에서는 면적을 절약하면서 속도를 높인 새로운 radix-4 ACS 구조를 제안하였다. 기존에 알려진 radix-4 고속 비터비 디코더를 위한 ACS연산을 rearranging 하여 덧셈기와 비교기의 개수를 줄이고 각 비교기에서 면적을 절약하였다. 또한 이로 인해 증가하는 ACSU에서의 임계경로 지연 시간의 문제를 경로 메트릭 메모리를 retiming하여 해결하였고, 제안된 ACS 구조의 특성을 이용하여 SDU에서 면적을 절약하였다. 실험을 통하여 제안된 구조가 FR4A의 구조와 비교 약 11%, CR4A의 구조와 비교 약 2%정도 AT product가 작은 것을 확인하여 비교된 ACS 구조 중 제안된 ACS 구조가 가장 효율적인 구조임을 확인할 수 있었다. 제안된 아키텍처가 FPGA에서 구현되고 실험되었기 때문에 custom design에서는 좀더 좋은 결과를 얻을 수 있으리라 기대된다.

참 고 문 헌

[1] A. P. Hekstra, "An alternative to metric rescaling in Viterbi decoders," IEEE Trans. Comm., Vol 37, No. 11, pp. 1220-1222, Nov. 1989.
[2] P. J. Black and T. H. Meng, "A 140-Mb/s 32-state radix-4 Viterbi decoder," IEEE J. Solid-State Circuits, Vol. 27, pp. 1877-1885, Dec. 1992.
[3] I. Kang and A. N. Willson, Jr., "A 0.24mW, 14.4Kbps, r=1/2, k=9 Viterbi decoder," CICC, 1997.
[4] Chi-Ying Tsui and Cheng, R.S.-K., Ling, C., "Low power ACS unit design for the Viterbi decoder," ISCAS, pp. 137-140, 1999.
[5] G. Fettweis and H. Meyr, "High-speed Viterbi processor: A systolic array solution," IEEE J. on Sel. Areas in Comm., Vol. SAC-8, pp. 1520-1534, Oct. 1990.
[6] I. Lee and J. L. Sonntag, "A new architecture for the fast Viterbi algorithm," IEEE Trans. Comm, Vol 51, No. 10, pp. 1624-1628, Oct. 2003.
[7] G. Fettweis, H. Meyr, "A 100Mbit/s Viterbi-decoder chip: Novel architecture and its realization," Proc. IEEE ICC, Vol. 2, pp.463-467, Aug. 1990.
[8] J. J. Kong and K. K. Parhi, "K-nested layered look-ahead method and architectures for high thoughput viterbi decoder," IEEE SIPS, pp. 99-104, Aug. 2003.
[9] P. J. Black and T. H. Meng, "1-Gb/s, four-state, sliding block Viterbi decoder," IEEE J. Solid-State Circuits, Vol. 32, pp. 797-804, June 1997.
[10] A. J. Viterbi and J. K. Omura, *Principles of digital communication and coding*, New York: McGraw-Hill, 1979.
[11] http://www.xilinx.com , Xilinx, Inc.

저 자 소 개

김 덕 환(학생회원)
2003년 서강대학교 컴퓨터학과 학사 졸업.
2004년 현재 서강대학교 컴퓨터학과 석사 과정.
<주관심분야 : CAD, VLSI 설계>

임 종 석(정회원)
1981년 서강대학교 전자공학과 학사 졸업.
1983년 한국과학기술원 전기 및 전자공학과 석사 졸업.
1989년 University of Maryland, College Park 전기공학과 박사 졸업.
1983년~1990년 8월 한국전자통신연구소 연구원.
1990년 9월~현재 서강대학교 컴퓨터학과 교수.
<주관심분야: 알고리즘, CAD, VLSI 설계>