논문 2008-45SD-11-13

상위수준 합성을 위한 비트단위 지연시간을 고려한 스케줄링

(Scheduling Considering Bit-Level Delays for High-Level Synthesis)

김 지 웅*, 신 현 철**

(Jiwoong Kim and Hyunchul Shin)

요 약

본 논문에서는 상위수준 합성에서의 비트단위 지연시간을 고려한 새로운 스케줄링 기법을 제안한다. 기존의 상위수준 합성을 위한 비트단위 지연시간 계산 방법은 특정 resource에서만 제한적으로 이용할 수 있었다. 하지만 본 연구에서는 다양한 resource에 대해서도 적용할 수 있는 효율적인 비트단위 지연시간 계산 방법을 개발하여, 이를 스케줄링에 적용하였다. 스케줄링 알고리즘은 리스트 스케줄링을 기반으로 하였으며, 스케줄링 과정에서 비트단위 지연시간을 고려하여 chaining을 수행한다. 또한 resource 제약조건하에서 성능을 더욱 향상시키기 위해 multi-cycle chaining을 수행할 수 있다. 잘 알려진 몇 가지 DSP 예제에 대한 실험 결과는 제안한 방법이 기존의 리스트 스케줄링에 비하여 평균 14.7% 성능을 향상시킬 수 있음을 보인다.

Abstract

In this paper, a new scheduling method considering bit-level delays for high-level synthesis is proposed. Conventional bit-level delay calculation for high-level synthesis was usually limited for specific resources. However, we have developed an efficient bit-level delay calculation method which is applicable to various resources, in this research. This method is applied to scheduling. The scheduling algorithm is based on list scheduling and executes chaining considering bit-level delays. Furthermore, multi-cycle chaining can be allowed to improve performance under resource constraints. Experimental results on several well-known DSP examples show that our method improves the performance of the results by 14.7% on the average.

Keywords : 비트단위 지연시간, chaining, 상위수준 합성, 스케줄링.

I. 서 론

VLSI의 복잡도 증가와 짧은 time-to-market으로 인한 productivity gap을 극복하기 위해 상위수준 합성, 설계 재사용, 플랫폼 기반 설계 등에 관한 여러 연구가 진행 되고 있다. 그 중 설계 추상화 단계를 register transfer level (RTL)에서 알고리즘 단계로 올리기 위한 상위수준 합성은 오랫동안 연구가 진행되었고, 학계나 산업체에서 여러 툴들이 개발 되었다. SPARK[1], xPilot[2], Forte Cynthesizer[3], Cyber[4]등의 툴들은 C/C++, SystemC와 같은 시스템 레벨 언어를 이용하여 HDL code를 생성하며, 설계를 최적화하기 위해 여러 가지 기법들을 사용하고 있다.

상위수준 합성 과정은 크게 스케줄링과 바인딩으로 이루어져 있으며 스케줄링은 주어진 제약조건 안에서 data flow graph (DFG)내의 각 operation (OP)의 시작 시간을 결정하는 과정으로 설계의 질을 결정하는데 가장 중요한 역할을 한다. 또한 설계의 성능을 향상시키기 위해 스케줄링 과정에서 여러 사이클에서 한 OP를 수행하는 multicycling이나 한 사이클에서 여러 OP를 수행하는 chaining 등을 사용한다[5].

기존의 상위수준 합성에서 시간 모델은 주로 계산의 복잡도를 줄이기 위해 resource의 최악의 경우 지연시

* 학생회원, 한양대학교 메카트로닉스공학과
(Mechatronics Engineering, Hanyang University)
** 평생회원, 한양대학교 전자전기제어계측공학과
(Electric Engineering and Computer Science, Hanyang University)
※ 본 논문은 지식경제부가 지원하는 국가 반도체 연구개발사업인 "시스템집적반도체기반기술개발사업(시스템 IC 2010)"을 통해 개발된 결과임을 밝힙니다.
접수일자: 2008년8월25일, 수정완료일: 2008년10월30일

간(worst case delay)로 모델링하였다. 따라서 chaining 시에는 두 OP의 최악의 경우 지연시간의 합이1 cycle 이내 일 때만 chaining이 가능했다. 그러나 실제적으로 많은 경우 chaining 했을 때의 최대 지연시간은 두 최악의 경우 지연시간의 합 보다 작은 값을 가지며, 따라서 하드웨어의 낭비가 발생한다. 이와 같은 낭비를 줄이기 위해 비트단위 chaining[6], 비트단위 분할 [7~8] 등의 기법이 제안되었다. 하지만 위 기법에서 사용한 지연시간 계산방법은 ripple carry adder와 같이 rippling effect를 가지는 특정 resource에서만 제한적으로 사용할 수 있고 여러 가지 복잡한 resource에서는 사용이 어려운 단점이 있다.

본 논문에서는 효율적이고 다양한 resource에 대해서도 적용할 수 있는 비트단위 지연시간 계산 방법을 개발하고 이를 리스트 스케줄링 과정에서 비트단위 chaining에 적용하였다. 또 resource 제약조건 하에서 latency를 더욱 최소화하기 위해 multi-cycle chaining을 수행 가능하도록 하였다.

II. 본 론

1. 비트단위 지연시간 기반 Chaining

기존의 상위수준 합성과정에서는 최악 경우의 지연시간 모델을 사용했기 때문에 chaining을 할 경우에 두 resource 각각의 최악의 경우 지연시간의 합으로 지연시간을 계산했다[5]. 그림 1은 두 개의 연속된 16bit ripple carry adder (ADD1, ADD2)의 비트단위 지연시간을 나타낸다. 그림 1의 오른쪽 그래프의 흰색 부분은 ADD1 (막대그래프의 윗부분), 검정색 부분은 ADD2의 수행을 각각 나타낸다. 덧셈기 한 개의 지연시간은 16δ 이지만 연속된 두 덧셈기는 rippling effect로 인해 17δ 가 걸림을 알 수 있다 (δ는 1bit 덧셈기의 지연시간).

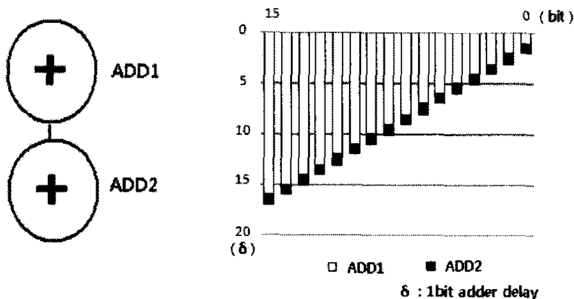

그림 1. 두 연속된 덧셈연산의 비트단위 지연시간
Fig. 1. Bit level delay of two chained adders.

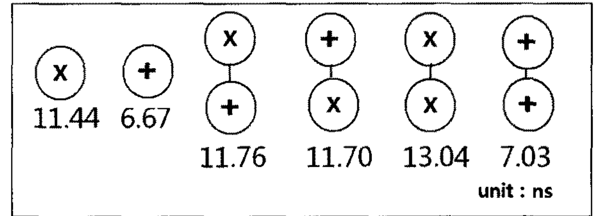

그림 2. 덧셈기와 곱셈기, chained OP들의 지연시간
Fig. 2. Delay values of an adder, a multiplier, and several chained operations.

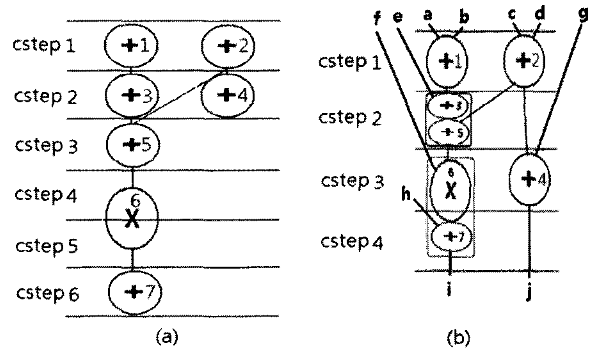

그림 3. 기존 리스트 스케줄링과 비트단위 지연시간을 고려한 스케줄링의 예
Fig. 3. Examples of list scheduling and scheduling considering bit level delays.

그림 2는 0.18um 공정에서 덧셈기와 곱셈기, 두 OP의 조합들의 지연시간을 나타낸다. 그림 1과 같은 원리로 두 OP를 chaining 했을 때 지연시간은 두 OP의 최악의 경우 지연시간의 합 보다 작은 값을 가지며 많은 slack이 존재함을 알 수 있다. 그림 3은 스케줄링 과정에서 비트단위 지연시간을 고려하여 chaining했을 경우 latency가 향상 될 수 있음을 나타낸다. 그림 3은 덧셈기 2 개, 곱셈기 1 개의 resource 제약조건을 가지고 스케줄링을 수행한 결과로 (a)는 기존 리스트 스케줄링의 결과를 나타내고 (b)는 비트단위 chaining을 적용한 스케줄링의 결과를 나타낸다. 그림 3 의 (b)의 경우 1 cycle의 주기가 두 OP를 chaining 했을 때 지연시간 보다 커야 되기 때문에 clock 주기가 약간 상승 (그림 2의 0.18um 공정에서 6.67 -> 7.03)하게 되지만 control step (cstep)의 감소 (6 -> 4)로 수행시간이 향상 (40.02 -> 28.12)됨을 알 수 있다. cstep 3, 4의 OP 6, 7은 multi-cycle chaining 되었다. 여기서 곱셈기 하나의 지연시간은 2 cycle이 걸리고 덧셈기는 1 cycle 걸리지만 비트단위 지연시간을 고려하면 곱셈기와 덧셈기를 2 cycle에 chaining 할 수 있다. 이때 cstep 4의 OP 7은 cstep 3의 OP 4와 동시에 사용되어 질 수 있

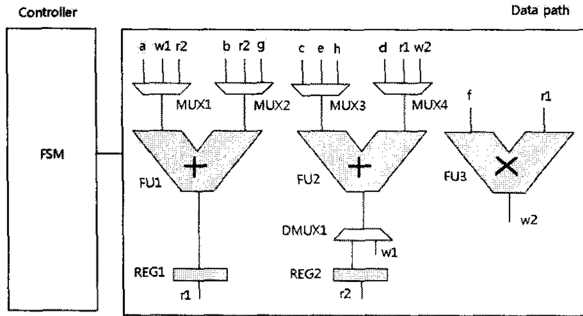

그림 4. 그림 3.(b)의 RTL 하드웨어 구조
Fig. 4. RTL architecture of Fig.3 (b).

표 1. 그림 3 예제의 로직 합성 결과
Table 1. Logic Synthesis Results of circuits in Fig. 3.

	그림 3 (a)	그림 3 (b)
Total area	55411.57	54429.59
%	100	98.2

기 때문에 resource 공유를 할 수 없다. 따라서 multi-cycle chaining은 resource가 증가하지 않도록 신중하게 적용되어야 한다. 그림 4는 그림 3(b) 스케줄링 결과의 OP 1, OP 4, OP 5 를 FU1에 OP 2, OP 3, OP 7을 FU2에 OP 6을 FU3에 바인딩 했을 때의 RTL 하드웨어 구조를 나타낸다. 여기서 chaining을 했을 때 추가의 interconnect 비용 (MUX, DMUX 등)이 발생함을 알 수 있다. 표 1은 그림 3 예제의 스케줄링 결과를 바인딩하여 Synopsys Design Compiler를 이용해 0.18um 공정에서 로직 합성한 결과이다. 비트단위 지연시간을 고려한 스케줄링 결과가 효율적인 바인딩을 통해 면적 오버헤드를 줄이거나 cstep 감소로 인한 controller의 면적 감소로 오히려 전체 면적이 줄어들 수 있음을 알 수 있다.

2. 비트단위 지연시간 계산

가. 기본개념

두 OP를 chaining했을 때 지연시간을 계산하기 위해 본 논문에서는 간단하면서 효율적인 방법을 제안한다. [8]에서는 지연시간을 계산하기 위해 1) addition followed by addition, 2) addition followed by multiplication, 3) multiplication followed by addition, 4) multiplication followed by multiplication으로 나누어 간단한 방법으로 계산을 하였다. 하지만 이와 같은 지연시간 계산 방법은 ripple carry adder와 같은 특정 resource에 제한적이고 다양하고 복잡한 resource에 사용될 수 없다. 이러한 단점을 극복하기 위해 본 논문에서는

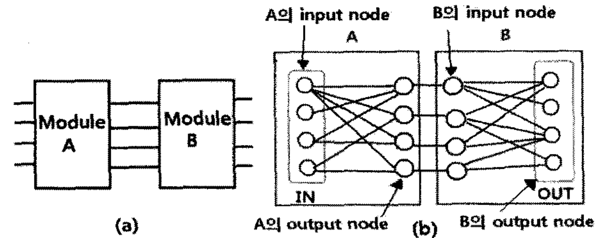

그림 5. 비트단위 지연시간 계산의 기본개념
Fig. 5. Basic concept of bit-level delay computation.

서는 timing library의 각각의 resource에 대해 pin-to-pin delay로 모델링하여 지연시간을 계산하였다. 그림 5는 두 개의 연속된 모듈을 chaining했을 때 지연시간을 계산하는 방법의 기본 개념을 나타내고 있다. 그림 5 (a)와 같이 모듈 A와 모듈 B가 연결되어 있을 때에, 이를 그림 5 (b)와 같이 그래프 형태로 나타낼 수 있다. 여기서 node는 각각의 pin을 나타내고 MUX나 bus같은 interconnect 지연시간을 weight로 갖고 있다. edge는 input pin과 output pin 사이에 지연시간 관계 (data path)를 나타내고 pin-to-pin 지연시간 (data path 지연시간)을 weight로 가지고 있다. 따라서 두 resource를 chaining 했을 때 지연시간은 그림 5 (b)의 IN에서 OUT까지의 가장 긴 경로의 weight합이 된다.

나. 지연시간 계산 알고리즘

비트단위 지연시간을 계산하기 위한 알고리즘은 그림 6과 같다. Cal_delay 함수는 두 OP (OP P, OP Q)를 chaining으로 연결 했을 때 비트단위 지연시간을 구하는 함수이고 내부에 Cal_module_delay 함수를 호출한다. Cal_module_delay 함수는 OP의 각 output node 에 대해 연결된 input node의 weight, edge의 weight, output node의 weight의 합들 중에 최대값을 output node의 누적 weight (outp->acc_weight)에 저장하게 된다. Cal_delay 함수의 step 1-4는 chaining했을 때 앞쪽의 OP (그림 5 (b)의 A부분)의 지연시간을 계산하는 부분으로 output_node에 이미 누적 weight가 계산 되어 있다면 수행되지 않는다. 예를 들어 연속된 세 OP인 OP1, OP 2, OP 3 을 chaining했을 때 지연시간을 계산할 경우 Cal_delay(1, 2), Cal_delay(2, 3)를 실행하게 된다. Cal_delay(1, 2)를 실행할 때는 OP 1의 output node에 누적 weight가 계산되지 않았기 때문에 step 2가 수행되지만 Cal_delay(2, 3)를 실행할 때는 OP 2의 output node에 weight가 Cal_delay(1, 2)에 의해 이미 계산되어 있으므로 step 2-3은 수행되지 않는다. step 6-8은 앞쪽

```
Cal_delay (OP P, OP Q)
1. if (OP P의 output node들의 acc_weight가 계산되어
      있지 않으면)
2.     OP P의 모든 node와 edge들의 weight 초기화
       (node의 weight = interconnect delay
        edge의 weight = data path delay)
3.     Cal_module_delay (OP P);
4. end if
5. OP Q의 모든 node와 edge들의 weight 초기화
6. for (모든 Q->input_node[i]에 대해)
7.     inp = Q->input_node[i];
8.     inp->weight = inp->prev_node->acc_weight
        + inp->weight;
9. end for
10. Cal_module_delay (OP Q);
11. total_delay = OP Q의 output node의
    acc_weight의 최대값
-----------------------------------------------
Cal_module_delay (OP R)
1. for (모든 R->output_node[j]에 대해)
2.     outp = R->output_node[j];
3.     for (모든 outp->in_edge[k]에 대해)
4.         temp_edge = outp->in_edge[k];
5.         temp_acc = outp->weight +
           temp_edge->weight + temp_edge->
           prev_node->weight;
6.         if (outp->acc_weight < temp_acc)
7.             outp->acc_weight = temp_acc;
8.         end if
9.     end for
10. end for
```

그림 6. 비트단위 지연시간 계산 알고리즘
Fig. 6. Bit level delay computation algorithm.

OP의 output node의 누적 weight (inp->prev_node->acc_weight)와 뒤쪽 OP의 input node의 weight (inp->weight)를 합산하는 부분이다 (그림 5 (b)에서 두 OP의 연결부분). step 10은 뒤쪽에 연결된 OP의 지연시간을 계산하여 step 11에서 가장 마지막 node들 (그림 5 (b)의 OUT으로 표시된 부분)중 가장 큰 누적 weight값이 두 OP를 chaining 했을 때 지연시간이 된다. 만약 Chaining할 OP구조가 그림 5의 (b)와 같다고 하면 Cal_delay 함수는 Cal_module_delay 함수를 호출하여 A부분의 delay를 계산하고 A의 output node 누적 weight (acc_weight)에 저장한다. 그리고 그 누적 weight를 B의 input node의 초기 weight와 합한 후 다시 Cal_module_delay를 호출하여 B의 output node에 누적 weight를 구하고 B의 누적 weight중 가장 큰 값이 최종 지연시간이 된다. 즉 Cal_delay 함수는 그림 5 (b) IN에서 OUT까지의 가장 긴 경로 (longest path)를 구하는 알고리즘이고, edge의 수에 따라 수행시간이 선형으로 증가한다.

3. 스케줄링 알고리즘

비트단위 지연시간을 고려한 스케줄링 알고리즘은 리스트 스케줄링을 기반으로 하였으며 그림 7과 같다. 그림 8은 그림 3과 동일한 DFG를 스케줄링 하는 과정으로 곱셈기 1개, 덧셈기 2개의 resource 제약조건을 가

```
1. 모든 node들의 우선순위 계산
2. Ready_list 생성
3. while (DFG내의 모든 OP가 schedule 될 때 까지)
4.     for (Ready_list에 있는 각각의 node n에 대해)
5.         Ready_list에서 chaining했을 때 지연시간이
           1 cycle 이내인 OP들을 chainable OP로 연결
6.     end for
7.     while (할당할 resource가 존재하고 Ready_list
        와 연결된 chain에 OP가 존재하면)
8.         candidates = Ready_list U { 선행 OP의
            scheduling이 끝난 chainable OP }
9.         candidates에서 우선순위가 가장 높은 OP를
           선택해 scheduling
10.    end while
11.    for (Ready_list내의 schedule된 모든 OP에 대해)
12.        if (multi-cycle chaining 수행 시 resource
             제약조건을 만족하면)
13.            multi-cycle chaining수행
14.        end if
15.    end for
16.    control step을 1 증가
17.    Ready_list를 업데이트
18. end while
```

그림 7. 스케줄링 알고리즘
Fig. 7. Scheduling algorithm.

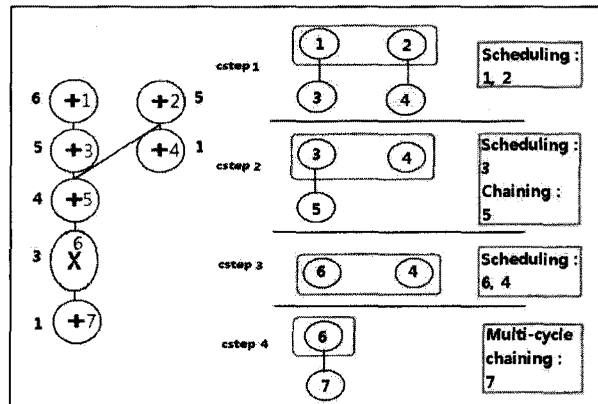

그림 8. 스케줄링 알고리즘의 예
Fig. 8. Example of scheduling algorithm.

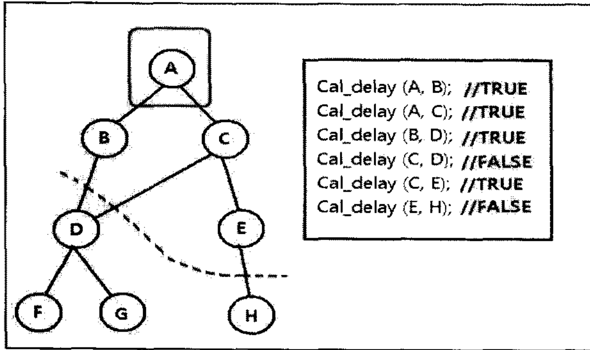

그림 9. 그림 7의 step 5를 설명하기 위한 예
Fig. 9. Example to explain step 5 in Fig. 7.

지고 스케줄링 한다. 그림 왼쪽 DFG의 OP밖의 번호는 곱셈기가 2 cycle, 덧셈기가 1 cycle 걸릴 때 해당 OP에서 output까지 경로 길이를 나타내며 이를 이용해 스케줄링 과정에서 OP들의 우선순위를 결정한다. 그림 8의 오른쪽 cstep 1에서 Ready_list를 생성하면 OP 1과 OP 2가 Ready_list에 삽입되고 (그림 7의 step 2) chaining 했을 때 지연시간이 1 cycle 이내인 OP 3과 OP 4가 chainable OP로 연결된다(step 4-6). 그중 우선순위가 가장 높은 OP 1이 스케줄링 되고 OP 2, OP 3은 우선순위가 같지만 하위 OP의 개수 (OP 2 : 5, OP 3 : 4)가 많은 OP 2가 스케줄링 된다(step 9). 하위 OP의 개수는 OP의 수행 cycle수 (곱셈기는 2, 덧셈기는 1)로 개수를 센다. 다음과 같은 과정이 DFG내의 모든 OP가 스케줄링 될 때 까지 반복된다. cstep 4에서 OP 6은 OP 7과 chaining 했을 때 resource 제약조건을 만족하기 때문에 (cstep 3에서 덧셈기 1개만 사용했고, cstep 4도 할당할 수 있는 덧셈기가 존재하므로) multi-cycle chaining을 수행한다(step 11-15). 스케줄링 알고리즘을 수행한 결과는 그림 3 (b)와 동일하다. 그림 9는 chaining 했을 때 1 cycle 이내에 걸리는 OP를 chainable OP로 연결하는 과정을 나타내는 또 다른 예이다. 그림 9의 왼쪽 DFG에서 Ready_list에 OP A가 있을 때 node들을 순회 하며 지연시간을 계산하게 된다. 그림 6의 Cal_delay함수를 이용한 계산순서는 그림 9의 오른쪽 박스 안과 같다. 여기서 주석의 TRUE는 1 cycle에 chaining이 가능함을 FALSE는 chaining 했을 때 1 cycle 이상 걸림을 나타낸다. 각 node를 A, B, C, E 순서로 순회하며 비트단위 지연시간을 계산하게 된다. 그림 10은 그림 9의 지연시간 계산 예의 세부과정을 나타낸다. OP D는 OP B와 지연시간 (OP A에서 OP B를 거쳐 OP D까지의 지연시간)은 1 cycle 안에 수행이 가능하고 OP C와의 지연시간 (OP A에서 OP C를 거쳐 OP D까지의 지연시간)은 1 cycle 안에 수행할 수 없다고 가정하면 OP D는 chainable OP에 연결시키지 않는다. 여기서 OP D의 하위 OP인 OP F와 OP G의 지연시간의 계산은 필요하지 않기 때문에 계산하지 않는다. 이와 같은 방법에 의해 OP A가 Ready_list에 있을 때 OP B, OP C, OP E가 1 cycle에 수행가능하며 chainable OP로 연결된다.

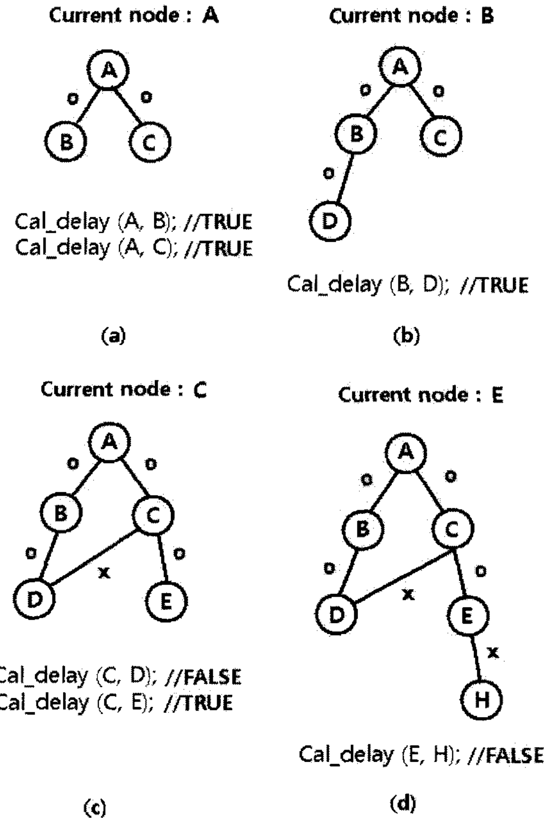

그림 10. 그림 9 의 수행 예
Fig. 10. Execution example of Fig. 9.

표 2. 스케줄링 결과
Table 2. Scheduling Results.

Bench -mark	# OP	Resource constraints	Execution time (ns)		Impro vement (%)
			LS	With chaining	
ewf	34	3+/-, 2*	120.06	91.39	22.2
dct	48	5+/-, 6*	53.36	49.21	7.7
wdf7	35	2+/-, 3*	106.72	98.42	7.8
iir7	29	3+/-, 3*	93.38	84.36	9.7
fir7	13	2+/-, 2*	60.03	56.24	6.3
fir11	21	2+/-, 3*	80.04	63.27	21.0
lattice	18	3+/-, 3*	73.37	49.21	32.9
volterra	27	2+/-, 3*	93.38	91.39	2.1
wavelet	40	4+/-, 3*	100.5	77.33	22.7

Ⅲ. 실 험 결 과

제안한 scheduling algorithm을 C language로 구현하여 실험 하였으며 180 um 공정 timing library를 이용하여 지연시간을 계산하였다. 표 2는 잘 알려진 여러 가지 DSP회로에 제안한 스케줄링 알고리즘을 적용한 결과이다. 표 2의 세 번째 column은 resource 제약조건을 나타내며, +/-는 덧셈/뺄셈을 수행하는 resource를, *는 곱셈을 수행하는 resource를 나타낸다. 네 번째 column은 기존의 list scheduling을 적용한 execution time (clock 주기 * clock cycle 수)을 나타내고 다섯 번째 column은 제안한 방법의 결과를 나타낸다. 마지막 column은 두 결과를 비교하여 제안된 방법의 향상도를 나타낸다. 결과는 제한한 방법이 평균 14.7% (최대 32.9%, 최소 2.1%)의 성능 향상이 있음을 보여준다.

Ⅳ. 결 론

본 논문에서는 상위수준 합성을 위한 새로운 비트단위 지연시간 계산기법과 이를 이용한 스케줄링 알고리즘을 제안하였다. 비트단위 지연시간을 계산하기 위해 resource 각각에 대해 pin-to-pin 지연시간이 모델링 된 timing library를 이용하여 간단한 최장 경로 알고리즘을 이용하여 chaining시 지연시간을 계산하였다. 스케줄링은 리스트 스케줄링을 기반으로 비트단위 지연시간을 이용해 chaining하고 resource 제약조건 안에서 multi-cycle chaining을 수행 가능하도록 하였다. 잘 알려진 몇 가지 DSP회로에 실험한 결과 평균 14.7%의 성능 향상이 있었다.

참 고 문 헌

[1] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. "SPARK: A Parallelizing Approach to the High-level Synthesis of Digital Circuits," Kluwer Academic Publishers, 2004.

[2] J. Cong, Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation," In Proc DAC, pp. 433 -438, 2006.

[3] Forte Design System Inc. San Jose, CA, "Cynthesizer," 2007 [Online] Available: http://www.forteds.com/

[4] K. Wakabayash, T. Okamoto, "C-Based Design Flow and EDA Tools: An ASIC and System Vendor Perspective," IEEE Trans. CAD, Vol. 19, no. 12, pp. 1507-1522, Dec. 2000.

[5] C. Hwang, J. Lee, and Y. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis," IEEE Trans. CAD, Vol. 10, no. 4, pp. 464-475, April 1991.

[6] S. Park and K. Choi, "Performance-driven high-level synthesis with bit-level chaining and clock selection," IEEE Trans. CAD, Vol. 2, no. 2, pp. 199 - 212, Feb. 2001.

[7] M. Molina, R. Ruiz-Sautua, J. Mendias, and R. Hermida, "Bitwise scheduling to balance the computational cost of behavioural specifications," IEEE Trans. CAD, Vol. 25, no. 1, pp. 31-46, Jan. 2006.

[8] R. Ruiz-Sautua, M. Molina, J. Mendias, "Exploiting Bit-Level Delay Calculations to Soften Read-After-Write Dependences in Behavioral Synthesis," IEEE Trans. CAD, Vol. 26, no. 9, pp. 1589-1601, Sep. 2006.

저 자 소 개

김 지 웅(학생회원)
2007년 한양대학교 전자컴퓨터 공학부 학사 졸업.
2007년~현재 한양대학교 메카트로닉스공학과 석사과정
<주관심분야 : CAD&VLSI, 저전력 설계, SoC 설계방법론>

신 현 철(평생회원)
1978년 서울대학교 전자공학과 학사 졸업.
1980년 한국과학기술원 전기 및 전자공학과 석사 졸업.
1983년~1987년 U.C. Berkeley 박사 졸업.
1983년~1987년 Fulbright scholarship
1987년~1989년 MTS, AT&T Bell Lab's, Murray Hill N.J., USA
1989년~현재 한양대학교 전자컴퓨터공학부 교수
1997년~현재 IDEC 한양대학교 지역센터 센터장
<주관심분야 : CAD&VLSI, 통신용 반도체 설계, 저전력 설계>