VHDL을 이용한 0-1 Knapsack 프로세서의 설계

이재진, 송호정, 송기용
충북대학교 컴퓨터공학과
ceicarus@just.chungbuk.ac.kr, hjsong@dce3.chungbuk.ac.kr, gysong@chungbuk.ac.kr

Design of the 0-1 Knapsack Processor using VHDL

Jae-Jin Lee, Ho-Jeong Song, and Gi-Yong Song
Dept. of Computer Engineering, Chungbuk National University

Abstract

The 0-1 knapsack processor performing dynamic programming is designed and implemented on a programmable logic device. Three types of a processor, each with different behavioral models, are presented, and the operation of a processor of each type is verified with an instance of the 0-1 knapsack problem.

I. 서론

특정 응용분야 전용의 작은 프로그램 가능한 시스템을 만드는 마이크로프로세서의 출현으로, PLD의 사용은 알고리즘을 하드웨어에 직접 구현하는 것을 가능하게 한다.

이 논문에서는, 동적 프로그래밍(dynamic programming)을 수행하는 하드웨어를 VHDL에 의해 설계하고 FPGA에 합성하였다.

조합형 최적화 문제(combinatorial optimization problem)들을 해결하기 위한 방식인 동적 프로그래밍은 최적화 문제가 최적 부-구조(optimal substruct)와 중복 부-문제(overlapping subproblem) 특성을 가지고 있다면 부-문제의 해답(solution)을 결합하여 문제를 해결한다. 만약 문제의 최적 해답(optimal solution)이 그 안에 부-문제에 대한 최적 해답들을 가지고 있다면 우리는 그 문제가 최적 부-구조를 가지고 있다고 말한다. 동적 프로그래밍이 적용되도록 하기 위해, 계속해서 새로운 부-문제를 만드는 방법보다는 오히려 같은 부-문제들을 계속에서 해결하려는 문제에 대하여 재귀 알고리즘을 적용한다. 재귀 알고리즘이 계속해서 같은 부-문제를 다시 만날 때 최적화 문제가 중복 부-문제(overlapping subproblem)를 가지고 있다고 말한다. 그러므로 다이나믹 프로그래밍의 방법은 기본적으로 buttom-up 방식이다. 동적 프로그래밍은 문제를 부-문제들을 공유하는 부-문제들로 나누고 계속해서 같은 문제들을 다시 만나면서 재귀적으로 부-문제들을 해결한다. 그런 다음 원형 문제들을 해결하기 위해 부-문제들의 해답을 결합한다. 모든 가능한 부-문제들(작은 부-문제들로 시작하는)이 각각 한번에 해결되고, 해결값이 테이블에 저장된다. 이것은 부-문제를 만날 때마다 해결 값(answer)을 다시 재계산하는 일을 피할 수 있도록 해준다.

II. 0-1 knapsack 문제

가장 전형적인 조합형 최적화 문제중의 하나인 knapsack 문제[2]를 고려해보고 knapsack 문제를 해결하는 프로세서를 구현해보자.

i가 1보다 크거나 같고 n보다 작거나 같은 경우 그리고 knapsack 용량(capacity)이 M인 경우 우리에게 이득(profit) p_i와 가중치(weight) w_i가 주어졌다고 가정해 보자. 이 문제는 $\sum x_i w_i \le M$, x_i는 0이거나 1, $1 \le i \le n$의 조건을 만족하는 $\sum x_i p_i$의 최대 값을 구하는 문제이다.

x_n의 값은 0또는1이다. 만약 x_n이 0이면 최대 이득(best profit)은 $1, \cdots, n-1$의 오브젝트 중의 어떤 값 중의 하나로 얻어질 수 있다. 만약 x_n이 1이면 $1, \cdots, n-1$의 오브젝트를 사용하여 최상의 이득을 얻음으로써 p_n의 이득을 가지게되고 남아있는 knapsack의 용량은 $M-w_n$이 된다.

우리는 이것을 아래의 점화 관계(recurrence relation)로 명시할 수 있다.

$0 \le m \le M$이고, $1 \le j \le n$일 때, 이득 $P(j,m)$는 오브젝트 $1, \cdots, j$, knapsack 용량 m을 사용한 0-1 knapsack 문제의 최적 해로 정의된다.

그러면, $2 \le j \le n$, $0 \le m \le M$인 경우

$P(j,m)=P(j-1,m)$ (만일 $w_j > m$)

$P(j,m)=\max\{P(j-1,m), Pj+P(j-1,m-w_j)\}$ (만일 $w_j \le m$)

이고, 초기 상태는

$P(1,m)=P_1$ (만일 $w_1 \le m$)

$P(1,m)=0$ (만일 $w_1 > m$)

이다.

최적 해는 P(n,M)에 의해서 주어진다.

최적 이득을 결정한 후에 우리는 이런 최적 해를 만드는 knapsack의 내용들을 테이블을 역추적 하면서 알아낼 수 있다. 이때 이 테이블은 최적 이득을 유도하는 계산의 연속된 내용을 저장하고 있다.

III. 0-1 knapsack 프로세서의 설계와 시뮬레이션

특별한 알고리즘 지향적인 프로세서는 두 가지 방법에 의해 설계될 수 있다. 목적 알고리즘에서 각 단계를 즉시 수행하는 특별한 구조와 전문적 명령을 가진 프로세서와 목적 알고리즘이 직접 구현되는 프로세서이다.

0-1 knapsack 문제를 해결하기 위한 동적 프로그래밍의 직접적인 하드웨어 구현이 이 논문에서 표현된다. 요구되는 출력과 최적 이득을 가진 knapsack의 내용을 결정하는 방법에 따라 3가지 종류의 knapsack 프로세서가 디자인되고 구현되었다.

각 타입의 프로세서는 리셋 신호가 입력되었을 때 적용된 칩의 입력 데이터에 의해 동작하고, 칩의 출력 핀에 의해 결과 값을 만들어 낸다.

1. Knapsack 프로세서 - 타입 1

타입 1의 knapsack 프로세서는 오직 최적 이득을 얻기 위하여 적용된다. 테이블에 저장된 최적 이득을 체크하는 방법에 의해 점화 관계를 처리하고 최적 이득을 결정하는 로직이 VHDL[3]에 의해서 설계된다.

타입 1 knapsack 프로세서의 코드는 그림 1에 보여진다.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity knapsack is
        port (
                weight_in, profit_in : in integer range 0 to 15;
                step1, step2, step3, step4, step5 : in std_logic;
                clk,rst : in std_logic;
                max_profit: out integer range 0 to 15
        );
end knapsack;

architecture knapsack_arc of knapsack is
procedure recurrence (      tem_profit :  in integer range 0 to 15;
                m_temp1 :       in integer range 0 to 15;
                m_temp2 :       in integer range 0 to 15;
                m_temp3 :       out integer range 0 to 15  ) is
variable temp1, temp2 : integer range 0 to 15;
begin
        temp1 := m_temp1;
        temp2 := tem_profit + m_temp2;
        if temp1  > temp2 then m_temp3 := temp1;
        else m_temp3 := temp2;
        end if;
end recurrence;

type matrix is array (0 to 15 ) of integer range 0 to 15;
signal matrix1 : matrix ;
signal matrix2 : matrix ;
signal matrix3 : matrix ;
signal matrix4 : matrix ;
type weightvec is array (1 to 5 ) of integer range 0 to 15;
signal weight : weightvec ;
type profitvec is array (1 to 5 ) of integer range 0 to 15;
signal profit : profitvec ;
signal i: integer range 1 to 5;
signal j: integer range 0 to 15;
signal done : std_logic;
begin
process (clk)
begin
        if clk='1' and clk'event then
                if step1='1' then
                        weight(1) <= weight_in;
                        profit(1) <= profit_in;
                elsif step2='1' then
                        weight(2) <= weight_in;
                        profit(2)  <= profit_in;
                elsif step3='1' then
                        weight(3) <= weight_in;
                        profit(3) <= profit_in;
                elsif step4='1' then
                        weight(4) <= weight_in;
                        profit(4) <= profit_in;
                elsif step5='1' then
                        weight(5) <= weight_in;
                        profit(5) <= profit_in;
                end if;
        end if;
end process;

process (clk,rst)
begin
        if rst='1' then
                j <=0;
                i <=1;
                done <='1';
        elsif clk='1' and clk'event then
                if done='1' then
                        if j=15 then
                                j <=0;
                                if i=5 then
                                        i <=1;
                                        done<='0';
                                else i <=i+1;
                                end if;
                        else j <= j+1;
                        end if;
                end if;
        end if;
end process;

process (i,j)
variable temp1,temp2, temp3, temp4 : integer range 0 to 15;
variable temp5,temp6, temp7, temp8 : integer range 0 to 15;
variable temp9,temp10, temp11, temp12 : integer range 0 to 15;
variable temp13,temp14, temp15, temp16 : integer range 0 to 15;
begin
        case i is
                when 1 =>
                        if  j < weight(1) then  matrix1(j)<=0;
                        else matrix1(j) <= profit(1);
                        end if;
                when 2 =>
                        if j < weight(2) then matrix2(j) <= matrix1(j);
                        else
                                temp1 := matrix1(j);
                                temp2 := matrix1 (j - weight(2));
                                temp3 := profit(2);
                                recurrence (temp3,temp1,temp2,temp4);
                                matrix2(j) <= temp4;
                        end if;
                when 3 =>
                        if j < weight(3) then matrix3(j) <= matrix2(j);
                        else
                                temp5 := matrix2(j);
                                temp6 := matrix2 (j - weight(3));
                                temp7 := profit(3);
                                recurrence (temp7,temp5,temp6,temp8);
                                matrix3(j) <= temp8;
                        end if;
                when 4 =>
                        if j < weight(4) then matrix4(j) <= matrix3(j);
                        else
                                temp9 := matrix3(j);
                                temp10 := matrix3 (j-weight(4));
                                temp11 := profit(4);
                                recurrence (temp11,temp9,temp10,temp12);
                                matrix4(j) <= temp12;
                        end if;
                when 5 =>
                        if 15 < weight(5) then
                                max_profit <= matrix4(15);
                        else
                                temp13 := matrix4(15);
                                temp14 := matrix4 (15 - weight(5));
                                temp15 := profit(5);
                                recurrence (temp15,temp13,temp14,temp16);
                                max_profit <= temp16;
                        end if;
                when others => null ;
        end case;
end process;
end knapsack_arc;
```

그림 1. 타입 1 knapsack 프로세서 VHDL 코드

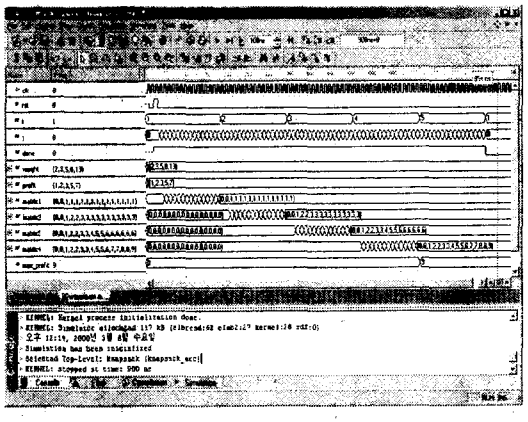

그림 2. 타입 1 knapsack 프로세서의 파형

각각의 가중치가 2,3,5,8,13이고 이득이

1,2,3,5,7, 용량이 15인 0-1 knapsack의 예로서, 타입 1 프로세서의 코드를 시뮬레이션한 결과 파형이 그림 2에 보여진다. 파형에서 볼 수 있는 배열 1,2,3,4는 각 p(j,m)의 값들을 보여주고 있는 이득 테이블(profit table)을 구성한다.

그림 3은 합성된 타입 1 프로세서를 Spartan VCS40-PQ240[4] FPGA에 구현한 것을 보여준다.

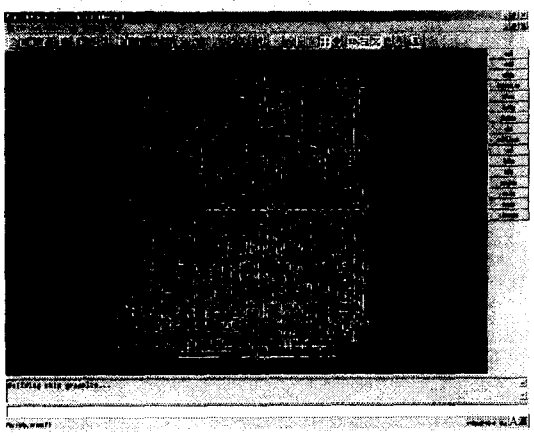

그림 3. 타입 1 knapsack 프로세서의 구현

2. Knapsack 프로세서 - 타입 2

최적 이득과 그것에 상응하는 내용들이 타입 2 knapsack 프로세서에 의해 결정된다. 이 경우에 최적 이득을 만드는 knapsack의 내용들에 대한 정보가 프로세서의 결과인 테이블을 연속적으로 역추적(tracing back) 하면서 얻어진다.

이득 테이블을 구성하는 배열을 동해 계산과정을 역추적 하는 코드가 그림 4에 보여진다.

각각의 가중치가 2,3,5,8이고 이득이 1,2,3,5, 용량이 10인 0-1 knapsack 프로세서에 대하여 각 내용의 정보를 포함하는 파형이 그림 5에 보여지고 FPGA에 구현된 것이 그림 6에 보여진다.

```
process (i)
variable temp9,temp10, temp11, temp12 : integer range 0 to 15;
begin
  case i is
      when 5 =>
          if max_profit = matrix3(10) then
              sel_item(3) <='0';
          else sel_item(3) <='1';
          end if;
      when 6=>
          if sel_item(3)='1' then
              if matrix3(10-weight(4)) = matrix2(10-weight(4)) then
                  sel_item(2) <='0';
              else sel_item(2) <='1';
              end if;
          else
              if matrix3(10) =matrix2(10)    then sel_item(2) <='0';
              else sel_item(2) <='1';
              end if;
          end if;
      when 7 =>
              if sel_item(3)='0' and sel_item(2)='0'  then
                  if matrix2(10) =matrix1(10)    then sel_item(1) <='0';
                  else sel_item(1) <='1';
                  end if;
              elsif sel_item(3)='0' and sel_item(2)='1'  then
                  if matrix2(10-weight(3)) =matrix1(10-weight(3)) then
                      sel_item(1) <='0';
                  else sel_item(1) <='1';
                  end if;
              elsif sel_item(3)='1' and sel_item(2)='0'  then
                  if matrix2(10-weight(4)) =matrix1(10-weight(4)) then
                      sel_item(1) <='0';
                  else sel_item(1) <='1';
                  end if;
              elsif sel_item(3)='1' and sel_item(2)='1'  then
              if matrix2(10-weight(4)-weight(3))
                =matrix1(10-weight(4)-weight(3)) then sel_item(1) <='0';
                  else sel_item(1) <='1';
                  end if;
          end if;
      when others => null;
  end case;
end process;
```

그림 4. 역추적 VHDL 코드

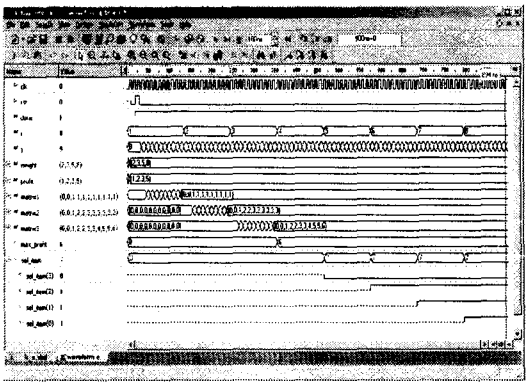

그림 5. 타입 2 knapsack 프로세서의 파형

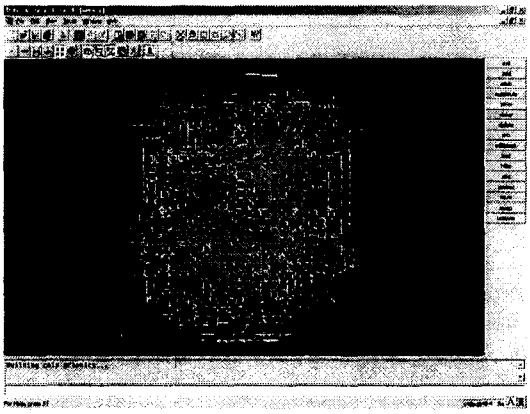

그림 6. 타입 2 knapsack 프로세서의 구현

3. Knapsack 프로세서- 타입3

타입3 프로세서 또한 최적 이득과 그것에 상응하는 내용들이 결정된다. 그러나 최적 이득을 가진 knapsack안의 내용들이 타입 2 프로세서의 경우처럼 프로세서의 결과를 통한 역추적이 아닌 내용의 조합 체크를 통해서 결정된다.

구현되었을 때, 예제의 범위가 작은 것으로 제한되었을 경우, 타입 3 knapsack 프로세서는 역추적 함수를 가진 타입 2 knapsack 프로세서 보다 더 적은 공간을 차지한다.

내용의 조합을 체크하는 코드가 그림 7에 보여진다.

```
process (i)
variable max_temp : integer range 0 to 15;
begin
        case i is
                when 5 =>
                        if max_profit = matrix3(15) then
                                sel_item(3) <='0';
                                max_temp:=max_profit;
                        else
                                sel_item(3) <='1';
                                max_temp := max_profit - profit(4);
                        end if;
                when 6 =>
                        if max_temp = 0  then
                                sel_item(2) <='0';
                                sel_item(1) <='0';
                                sel_item(0) <='0';
                        elsif max_temp = profit(3)  then
                                sel_item(2) <='1';
                                sel_item(1) <='0';
                                sel_item(0) <='0';
                        elsif max_temp = profit(2)  then
                                sel_item(2) <='0';
                                sel_item(1) <='1';
                                sel_item(0) <='0';
                        elsif max_temp = profit(1)  then
                                sel_item(2) <='0';
                                sel_item(1) <='0';
                                sel_item(0) <='1';
                        elsif max_temp = profit(3) + profit(2)  then
                                sel_item(2) <='1';
                                sel_item(1) <='1';
                                sel_item(0) <='0';
                        elsif max_temp = profit(3) + profit(1)  then
                                sel_item(2) <='1';
                                sel_item(1) <='0';
                                sel_item(0) <='1';
                        elsif max_temp = profit(2) + profit(1)  then
                                sel_item(2) <='0';
                                sel_item(1) <='1';
                                sel_item(0) <='1';
                        elsif max_temp=profit(3)+profit(2)+profit(1)  then
                                sel_item(2) <='1';
                                sel_item(1) <='1';
                                sel_item(0) <='1';
                        end if;
                when others => null;
        end case;
end process;
```

그림 7. 조합 체크 VHDL 코드

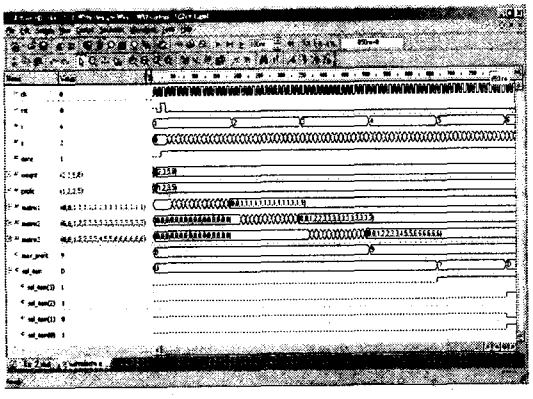

그림 8. 타입 8 knapsack 프로세서의 파형

각각의 가중치가 2,3,5,8, 이득이 1,2,3,5, 용량이 22인 경우 각 내용에 대한 정보를 포함하는 파형이 그림 8에 보여진다. 그리고 FPGA에 구현된 것이 그림 9에 보여진다.

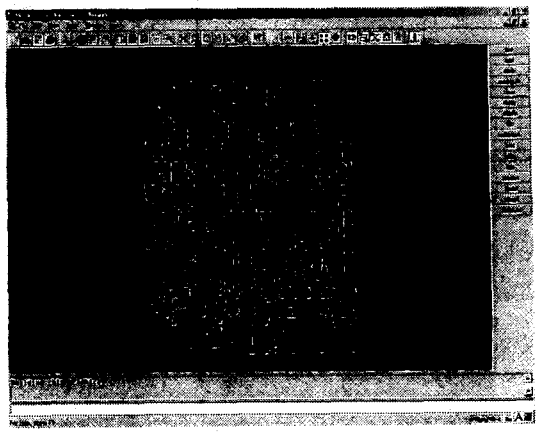

그림 9. 타입 3 knapsack 프로세서의 구현

IV. 결론

3종류의 0-1 knapsack 프로세서가 VHDL에 의해 설계되고 FPGA에서 합성되었다. 이 논문에서 표현된 특정 어플리케이션을 위한 하드웨어 디자인은 알고리즘 영역 공간 으로부터 레이아웃 영역으로의 직접적인 매핑으로 보여질 수 있고 밀집된 결과를 필요로 하는 최종 합성의 결과를 낼 수도 있을 것이다.

또한 전용 명령어를 가진 프로세서로 설계하는 경우 더 밀집된 레이아웃으로 이끌 수 있을 것이다.

참고문헌

[1] Cormen, T.H., Leiserson, C.E., and Rivest, R.L. *Int. to Algorithms*, McGraw Hill.

[2] Stinson, D.R., *An Int. to the Design and Analysis of Algorithms*, 2nd Ed, The Charles Babbage Research Centre.

[3] Smith, D.J., *HDL Chip Design*, Doone Publications.

[4] Xilinx, 1999 *Xilinx Data Book.*