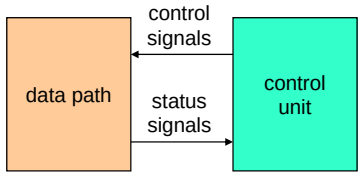
Chapter 8

데이터 경로와 제어장치 및 산술연산 회로

8.1 데이터 경로와 제어장치

- 데이터경로와 제어장치
  - 많은 순차 회로의 설계는 다음의 두 부분으로 구성
    - datapath: data의 이동 및 연산을 위한 장치
      control unit에 상태신호 제공
    - control unit: datapath에서 적절한 순서로 data 이동 및 연산을 수행할 수 있도록 제어신호 제공.
  - 먼저, datapath를 설계
  - 다음에, control unit를 설계



8.2 수의 표현

- 2진수 표현: $B = b_{n-1}b_{n-2} \ldots b_1b_0$
  - unsigned integer:

    값 = unsigned(B) = $\sum_{i=0}^{n-1} b_i 2^i$

  - signed integer (2's complement 표기법):

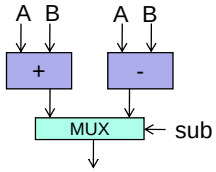
    값 = signed(B) = $-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$

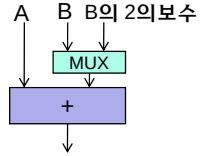
- (예) B = 10100010
  - unsigned(B) = $2^7 + 2^5 + 2^1 = 128 + 32 + 2 = 162$
  - signed(B) = $-2^7 + 2^5 + 2^1 = -128 + 32 + 2 = -94$ 또는
    $= -2^8 + (2^7 + 2^5 + 2^1) = -256+162 = -94$

    (참고) 10100010 의 2의 보수
    $= 01011110 = 2^6+2^4+ 2^3+2^2+2^1 = 64+16+8+4+2 = 94$

8.3 가감산기와 shifter

- 가산기 설계
  - 방법1: 논리회로를 직접 설계
  - 방법2: + 연산자 이용
  - FPGA를 사용한 설계에서는 + 연산자를 이용하는 것이 좋음
    → 합성도구가 carry 전용 경로를 사용하므로 더 효율적인 합성을 수행
- 가감산기 설계
  - 가산기, 감산기 별도 구현
  - 2의 보수 덧셈을 사용하여 뺄셈 수행



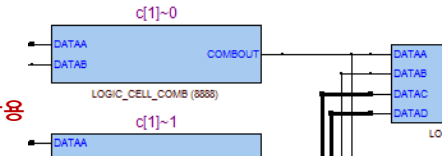


가산기 : 논리회로 직접 설계

```
module adder(data1, data2, cin, sum, cout);
    input  [3:0] data1, data2;
    input  cin;
    output [3:0] sum;
    output cout;

    integer i;
    reg [4:0] c;

    assign sum = data1 ^ data2  ^ c[3:0];
    assign cout = c[4];

    always @* begin
        c[0] = cin;
        for (i=0; i<4; i=i+1) begin
            c[i+1] = data1[i] & data2[i] | data1[i] & c[i] | data2[i] & c[i];
        end
    end

endmodule
```

합성결과

Total logic elements	10
Total combinational functions	10



LE당 출력 1개 사용

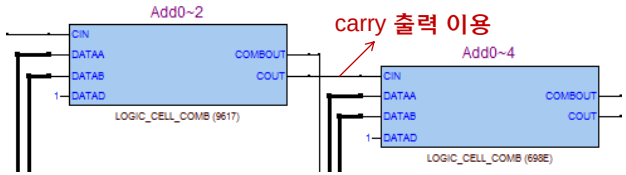
가산기 : + 연산자 이용

```
module adder(data1, data2, cin, sum, cout);
    input  [3:0] data1, data2;
    input  cin;
    output [3:0] sum;
    output cout;

    assign {cout, sum} = data1 + data2 + cin;

endmodule
```

합성결과

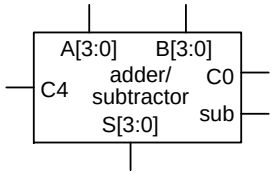
Total logic elements	6
Total combinational functions	6

더 효율적



가감산기

- 가감산기 동작
  - sub=0일때 가산기
    - C0, C4는 carry 입출력
  - sub=1일때 감산기
    - C0, C4는 borrow 입출력



- 가감산기 설계 – 가산기와 감산기 이용

```
module addsub(sub, a, b, cin, s, cout);
   input sub;
   input [3:0] a, b;
   input cin;
   output [3:0] s;
   output cout;

   assign {cout, s} = sub ? (a - b - cin) : (a + b + cin);
endmodule
```

가산기, 감산기 개별적 구현 가능성

가감산기(2)

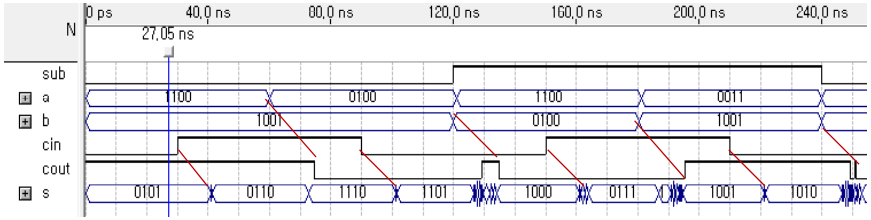
- 가감산기 설계 - 뺄셈을 2의 보수 덧셈으로 구현
  - 2의 보수 = (1의 보수) + 1
  - borrow=0이면 A + (B의 1의 보수) + 1
    borrow=1이면 A + (B의 1의 보수) + 0
  - borrow출력은 가산기의 carry출력과 반대가 됨

```
module addsub(sub, a, b, cin, s, cout);
    input sub;
    input [3:0] a, b;
    input  cin;
    output [3:0] s;
    output cout;

    wire [3:0] b2;
    wire cin2, cout2;

    assign b2 = sub ? ~b : b;
    assign cin2 = sub ? ~cin : cin;
    assign {cout2, s} = a + b2 + cin2;
    assign cout = sub ? ~cout2 : cout2;
endmodule
```

가산기

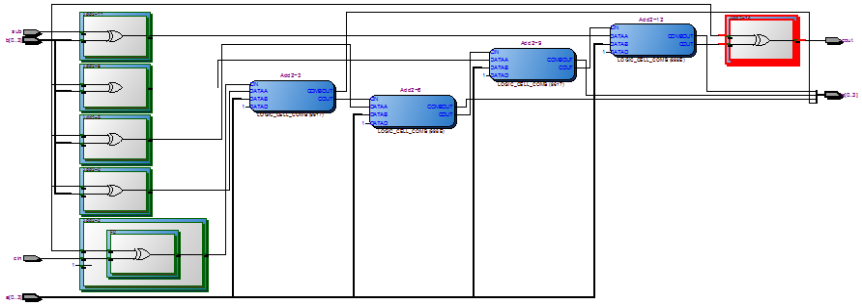
가감산기(3)

- simulation (두 설계가 같음)



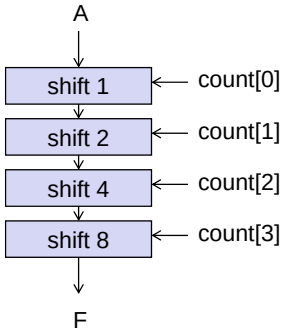
가감산기(4)

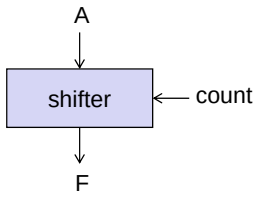
- 합성결과



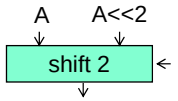
Shifter

- Barrel Shifter
  - 한 번에 여러 비트(가변)를 이동시킬 수 있는 조합회로 (shift 횟수=count)
- 구현





각 shift회로는
2x1 multiplexer로 구현



- 시프트 연산자 사용: A << count → 합성도구가 구현

8.4 곱셈기

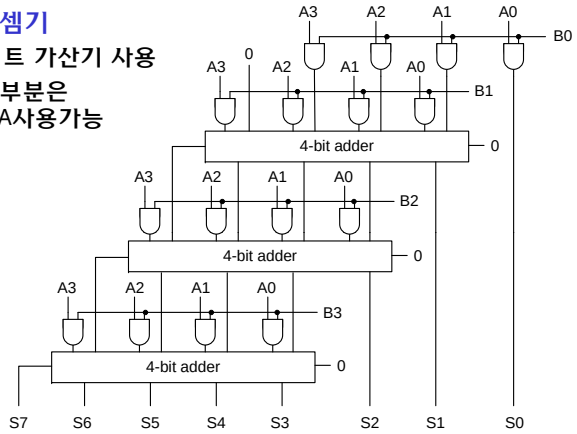
- 2진수 곱셈 (unsigned)

```
   1101      Multiplicand (13)
x) 1011      Multiplier(11)
   1101
  1101
 0000
1101
10001011     Product(143)
```

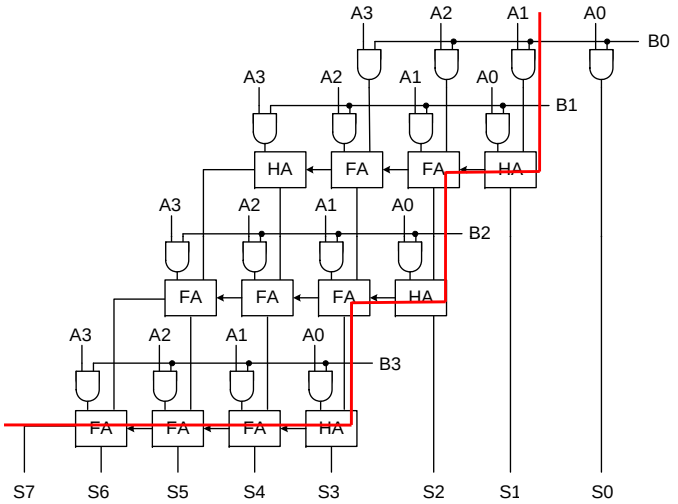
- 곱셈기 설계
  - 조합회로 설계: 배열 곱셈기
    - n-1개의 n-bit 가산기를 사용
  - 순차회로 설계:
    - 1개의 n-bit 가산기를 사용
    - 한번에 한 자리씩 곱셈 수행 → n번 수행

배열 곱셈기

- n비트 배열 곱셈기
  - n-1개의 n비트 가산기 사용
  - 0을 더하는 부분은 FA대신에 HA사용가능

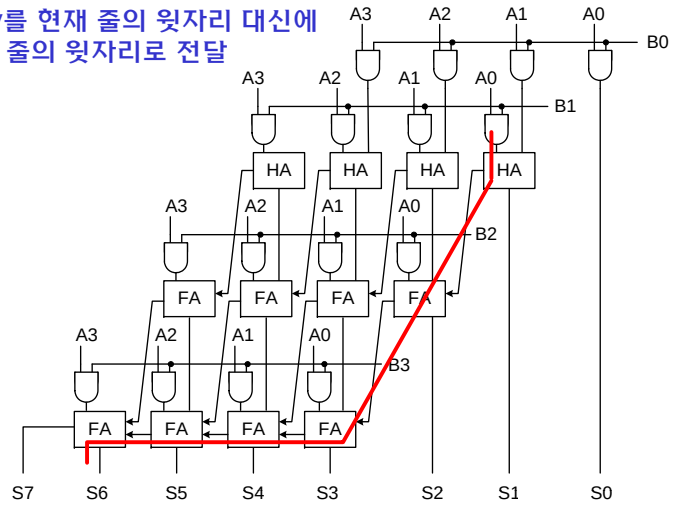


- 곱셈연산자 사용 : A * B → 합성도구가 조합회로 곱셈기 구현



Carry Save 배열 곱셈기

- carry를 현재 줄의 윗자리 대신에 다음 줄의 윗자리로 전달



8.5 순차 곱셈기

- 2진수 곱셈 과정

```
    1101      Multiplicand
x)  1011      Multiplier
    ----
    1101
   1101       --------  Shift Left Multiplicand
  100111
  0000
 100111
1101
----
10001111      Product
```

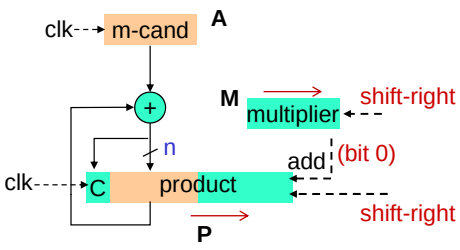
- 순차곱셈기
  - 한 번에 multiplier 1자리씩 곱셈 계산
  - datapath 와 control unit 로 구성됨
    - datapath를 설계하면서 필요한 제어신호 정의
    - control unit에서 적절한 순서로 제어신호를 공급

Datapath

- Algorithm

  M = multiplier (n bit)
  A = multiplicand (n-bit)
  P = 0 (2n-bit)
  repeat *n* times **do**
  if (M[0]=1) $P_H = P_H + A$
  shift right M
  shift right P
  end

Datapath



P_H는 P의 upper half

- 특징
  - multiplicand를 shift-left하는 것 대신에 product를 shift-right함
  - n-bit adder 및 n-bit multiplicand register 사용


```
module Datapath (product, m0, word1, word2, Load, Shift, Add, reset, clock);
    output [7:0] product;
    output m0;
    input [3:0]  word1, word2;
    input Load, Shift, Add, clock, reset;
    reg [7:0]  product;
    reg [3:0]  multiplier, multiplicand;
    reg c_out;
    wire m0;

    assign m0 = multiplier[0];

    always @ (posedge clock or posedge reset) begin
        if (reset) begin
            multiplicand <= 0; multiplier <= 0; product <= 0;
        end else if (Load) begin
            multiplicand <= word1; multiplier <= word2; product <= 0;
        end else if (Shift) begin
            multiplier <= multiplier >> 1;
            {c_out, product} <= {c_out, product} >> 1;
        end else if (Add) begin
            {c_out, product[7:4] } <= product[7:4] + multiplicand;
        end
    end
endmodule
```

word1 : multiplicand
word2 : multiplier

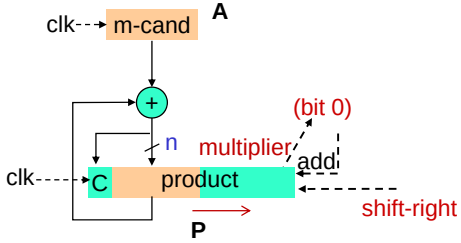
control unit에서 제어신호를 공급

Datapath (product와 multiplier를 같이 사용)

- 수정사항
  - Product의 하위부분에 multiplier 저장
    → multiplier 레지스터 불필요
- Algorithm

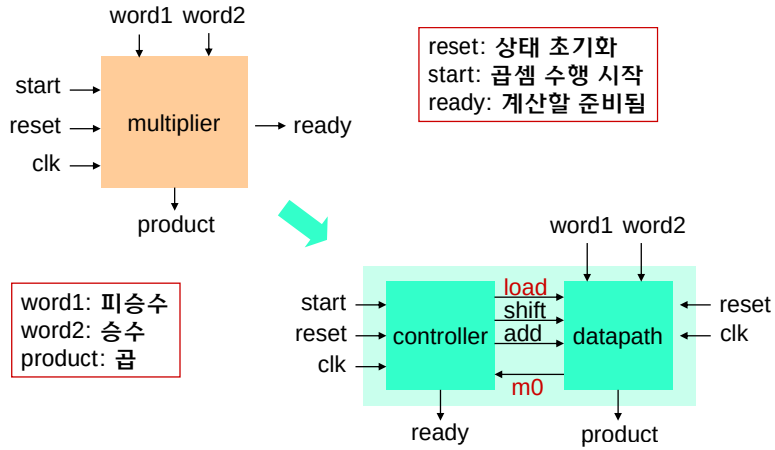
  A = multiplicand (n-bit)
  P_L = multiplier
  P_H = 0 (P: 2n-bit)
  repeat *n* times **do**
  if (M[0]=1) $P_H = P_H + A$
  ~~shift right M~~
  shift right P
  end

Datapath 4



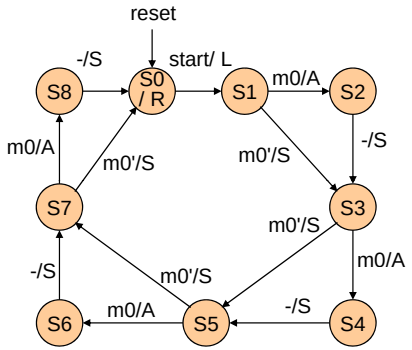
순차 곱셈기

- 블록 다이어그램과 내부 구조도



순차 곱셈기

- controller의 상태도



L: Load
R: Ready
A: Add
S: Shift

Load: 입력을 레지스터에 저장하는 제어신호

shift 또는 add-shift 동작을 반복

in/out 은
  in=1일 때 out=1이 됨을 표시
in'/out은
  in=0일 때 out=1이 됨을 표시
표시되지 않은 출력은 0을 표시

순차곱셈기

- Verilog description

```
module multiplier1(product, Ready, word1, word2, Start, reset, clock, state);
    output [7:0] product;
    output Ready;
    input [3:0] word1, word2;
    input Start, clock, reset;
    output [3:0] state;              // for observation
    wire m0, Load, Shift, Add;

    Datapath u1 (product, m0, word1, word2, Load, Shift, Add, reset, clock);
    Controller u2 (Load, Shift, Add, Ready, m0, Start, reset, clock, state);
endmodule
```

Datapath는 앞의 정의 참조


```
module Controller (Load, Shift, Add, Ready, m0, Start, reset, clock, state);
    output Load, Shift, Add, Ready;
    input m0, Start, clock, reset;
    output [3:0] state;          // for observation
    reg [3:0] state, next_state;
    parameter S0 = 0, S1 = 8, S2 = 9, S3 = 10, S4 = 11,
              S5 = 12, S6 = 13, S7 =14, S8 = 15;
    reg Load, Shift, Add;
    wire Ready;

    // State transition
    always @ (posedge clock or posedge reset) begin
        if (reset) state <= S0;
        else state <= next_state;
    end
```

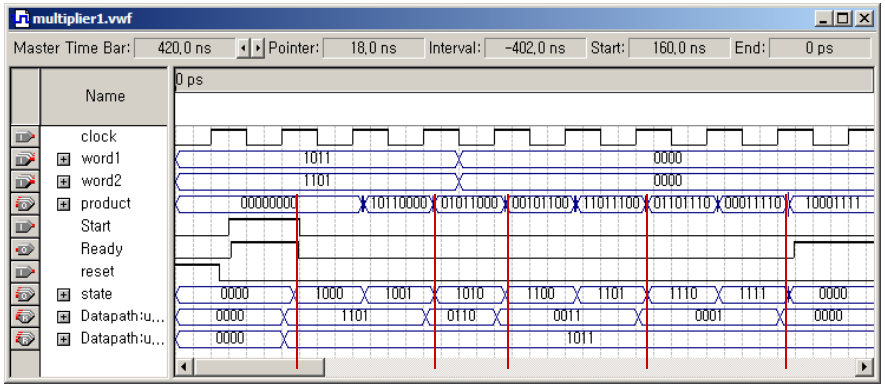
초기상태와 곱셈이 진행되는 8개 상태를 최상위비트로 구분하도록 상태 인코딩함

(계속)


```
    // Next state and control output logic
    always @ (state or Start or m0) begin
        Load = 0;  Shift = 0; Add = 0;
        case (state)
          S0:   if (Start) begin Load = 1; next_state = S1; end
                else next_state =  S0;
          S1:   if (m0) begin Add = 1; next_state = S2; end
                else begin Shift = 1; next_state = S3; end
          S2:   begin Shift = 1; next_state = S3; end
          S3:   if (m0) begin Add = 1; next_state = S4; end
                else begin Shift = 1; next_state = S5; end
          S4:   begin Shift = 1; next_state = S5; end
          S5:   if (m0) begin Add = 1; next_state = S6; end
                else begin Shift = 1; next_state = S7; end
          S6:   begin Shift = 1; next_state = S7; end
          S7:   if (m0) begin Add = 1; next_state = S8; end
                else begin Shift = 1; next_state = S0; end
          S8:   begin Shift=1; next_state = S0; end
          default: next_state = S0;
        endcase
    end

    assign  Ready = (state==S0) && ~reset;
endmodule
```

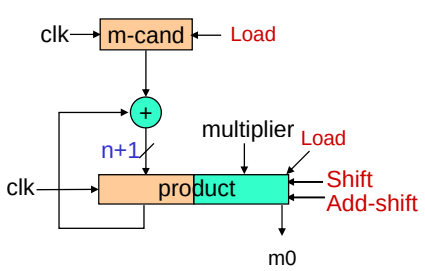
순차 곱셈기

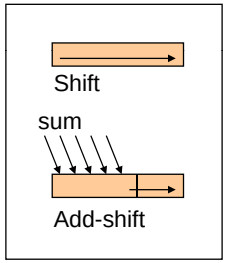
- Simulation
  - 13 x 11 = 143



8.5 빠른 곱셈기

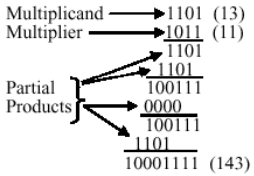
- datapath
  - multiplier를 product의 하위부분을 공유 사용
  - add 결과 저장과 shift를 한 번에 수행 → 동작: shift 또는 add-shift

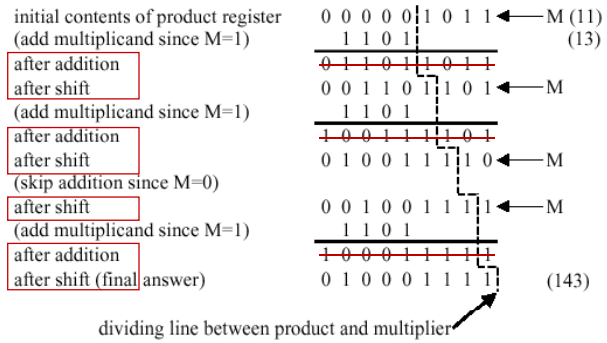




빠른 곱셈기

- 곱셈 수행 과정





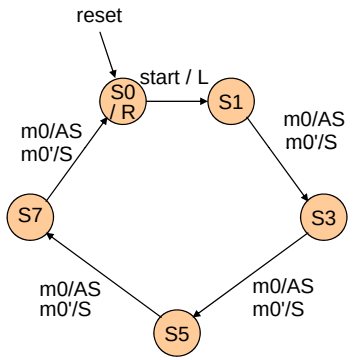
빠른 곱셈기

```
module Datapath (product, m0, word1, word2, Load, Shift, Add_Shift, reset, clock);
    output [7:0] product;
    output m0;
    input [3:0] word1, word2;
    input Load, Shift, Add_Shift, clock, reset;
    reg [7:0] product;
    reg [3:0] multiplier, multiplicand;
    wire [4:0] sum;          // 5-bit

    assign m0 = product[0];
    assign sum = product[7:4] + multiplicand;
    always @ (posedge clock or posedge reset) begin
        if (reset) begin multiplicand <= 0; product <= 0; end
        else if (Load) begin
            multiplicand <= word1;
            product <= {4'b0, word2};
        end else if (Shift) begin
            product <= product >> 1;              // product <= {1'b0, product[7:1]};
        end else if (Add_Shift) begin
            product[7:3] <= sum;
            product[2:0] <= product[3:1];         // product <= {sum, product[3:1]};
        end
    end
endmodule
```

빠른 곱셈기

- controller의 상태도

L: Load
R: Ready
S: Shift
AS: Add-Shift



빠른 곱셈기

```
module Controller (Load, Shift, Add_Shift, Ready, m0, Start, reset, clock, state);
    output Load, Shift, Add_Shift, Ready;
    input m0, Start, clock, reset;
    output [2:0] state;
    reg [2:0] state, next_state;
    parameter  S0 = 0, S1 = 4, S3 = 5, S5 = 6, S7 = 7;
    wire Load, Shift, Add_Shift, Ready;

    always @ (posedge clock or posedge reset) // State transitions
        if (reset) state <= S0;
        else state <= next_state;
```

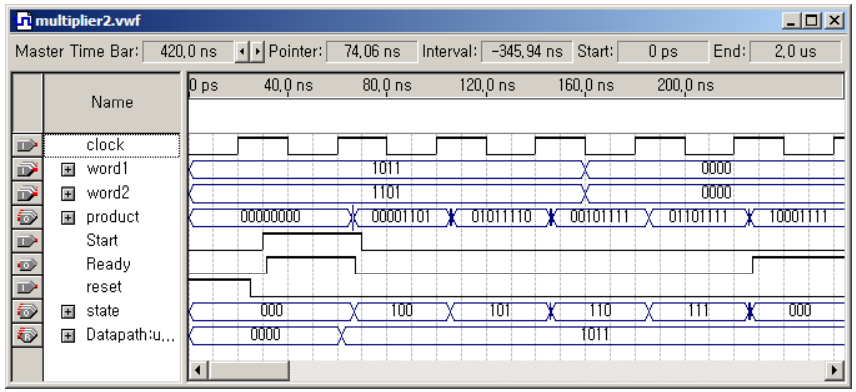
(계속)

빠른 곱셈기

```
    always @ (state or Start or m0) begin  // Next state logic
        case (state)
          S0:   if (Start) next_state = S1;  else next_state = S0;
          S1:   next_state = S3;
          S3:   next_state = S5;
          S5:   next_state = S7;
          S7:   next_state = S0;
          default: next_state = S0;
        endcase
    end
    // outputs
    assign Add_Shift = (state==S1 || state==S3 || state==S5 || state==S7) & m0;
    assign Shift = (state==S1 || state==S3 || state==S5 || state==S7) & ~m0;
    assign Load = (state==S0) && Start;
    assign Ready = (state==S0) && ~reset;
endmodule
```

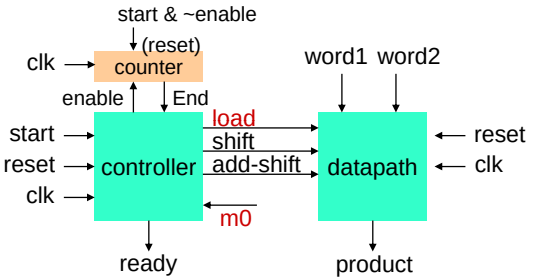

빠른 곱셈기

- Simulation



8.6 카운터를 사용한 곱셈기 제어

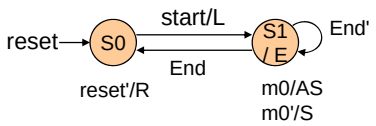
- 곱셈기 제어의 문제점 및 해결책
  - 문제점: 비트 수를 변경하면 제어기를 수정해야 함
  - 해결책: 카운터를 사용하여 제어기 설계



counter 동작
곱셈 시작할 때에 count ← 0
각 곱셈 단계에서 count 증가
counter 출력이 N-1이면 End 신호 출력

카운터를 사용한 곱셈기 제어

- 상태도



L: Load
R: Ready
S: Shift
AS: Add-Shift
E: Enable (counter)

- Verilog description
  - counter

```
module counter(count, Enable, reset, clock);
    output [3:0] count;
    input Enable, reset, clock;
    reg [3:0] count;

    always @(posedge clock)
        if (reset) count <= 0;
        else if (Enable) count <= count + 1;
endmodule
```


카운터를 사용한 곱셈기 제어

```
module multiplier3 (product, Ready,  word1, word2, Start, reset,
      clock);
    parameter N=4;
    output [2*N-1:0]  product;
    output Ready;
    input [N-1:0] word1, word2;
    input Start, clock, reset;
    wire m0, Load, Shift, Add_Shift, Enable, End;
    wire [3:0] count;

    Datapath u1 (product, m0, word1, word2, Load, Shift, Add_Shift,
                 reset, clock);
    Controller u2 (Load, Shift, Add_Shift, Enable, Ready, m0, Start,
                   End, reset, clock);
    counter u3 (count, Enable, Start & ~Enable, clock);
    assign End = (count==N-1);
endmodule
```

counter reset 신호

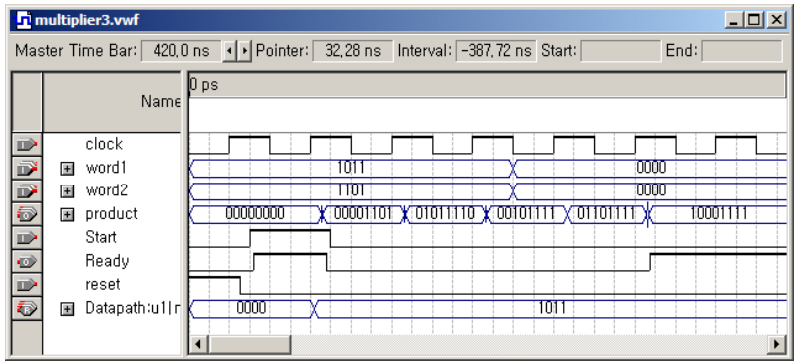
datapath는 앞의 빠른 곱셈기와 같음

카운터를 사용한 곱셈기 제어

```
module Controller (Load, Shift, Add_Shift, Enable, Ready, m0, Start,
                   End, reset, clock);
    output Load, Shift, Add_Shift, Enable, Ready;
    input m0, Start, End, clock, reset;
    reg state, next_state;
    parameter S0 = 0, S1 = 1;
    wire Shift, Add_Shift, Load, Ready;

    always @ (posedge clock or posedge reset)   // State transitions
        if (reset) state <= S0; else state <= next_state;
    always @ (state or Start or m0)     // Next state and control logic
        case (state)
          S0:  if (Start) next_state = S1; else next_state = S0;
          S1:  if (End) next_state = S0; else next_state = S1;
        endcase
    assign Add_Shift = (state==S1) & m0;
    assign Shift = (state==S1) & ~m0;
    assign Load = (state==S0) && Start;
    assign Enable = (state==S1);
    assign Ready = (state==S0) && ~reset;
endmodule
```

카운터를 사용한 곱셈기 제어

- Simulation (N=4)



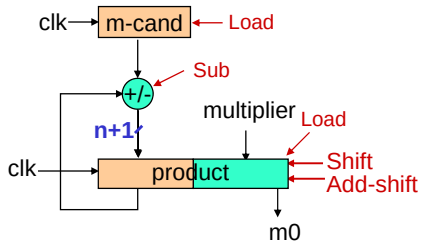
8.7 signed 곱셈기

- Binary notation: B = $b_{n-1}b_{n-2} \ldots b_1b_0$
  - unsigned(B) = $\sum_{i=0}^{n-1} b_i 2^i$
  - signed(B) = $-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$

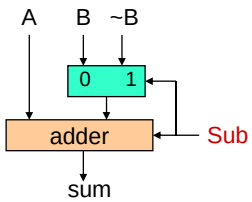
- unsigned 곱셈기를 signed 곱셈기로 간단하게 수정하는 방법
  - 최상위 bit가 1이면 multiplicand를 더하는 것 대신에 뺄셈을 수행
  - 뺄셈을 수행할 수 있도록 datapath를 수정→ 2의 보수 덧셈 (가감산기)
  - adder를 signed number에 대한 덧셈/뺄셈에 사용할 때에는 carry-out은 사용하지 않음 (대신에 n+1비트 가감산수행)
  - product를 shift-right 시킬 때에 부호(최상위 bit)를 그대로 유지해야 함

signed 곱셈기

- datapath
  - n+1 비트 가감산기 사용
    - multiplicand는 n+1비트로 부호확장
  - carry출력은 사용하지 않음

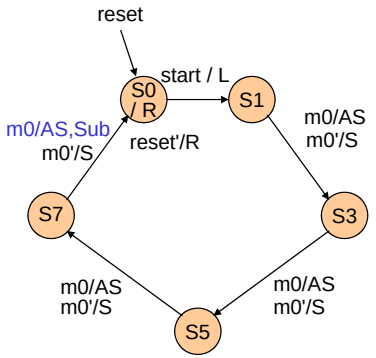


adder/subtractor



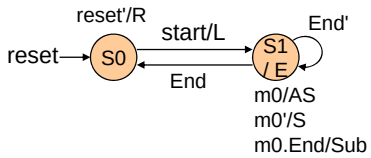
signed 곱셈기

- controller의 상태도



L: Load
R: Ready
S: Shift
AS: Add-Shift
Sub: Sub

카운터 사용한 상태도



signed 곱셈기

```
module Datapath (product, m0, word1, word2, Load, Shift, Add_Shift, Sub,
                 reset, clock);
  output [7:0] product;
  output m0;
  input [3:0] word1, word2;
  input Load, Shift, Add_Shift, Sub, clock, reset;
  reg [8:0] eproduct;          // 9-bit
  reg [4:0] multiplicand;      // 5-bit
  wire[4:0] sum;               // 5-bit

  assign product = eproduct[7:0];
  assign m0 = eproduct[0];
  always @ (posedge clock or posedge reset) begin
       if (reset) begin multiplicand <= 0; eproduct <= 0; end
       else if (Load)  begin
              multiplicand <= {word1[3],word1}; // sign-extension
              eproduct <= {5'b0, word2};
       end else if (Shift)     // signed shift-right
              eproduct <= {eproduct[8], eproduct[8:1]};
    else if (Add_Shift)        // signed add & shift right
              eproduct <= {sum[4], sum, eproduct[3:1] };
  end
  assign sum = eproduct[8:4] + (Sub ? ~multiplicand : multiplicand) + Sub;
endmodule
```

adder/subtractor (sub=1일때 뺄셈)

signed 곱셈기

```
module Controller (Load, Shift, Add_Shift, Sub, Enable, Ready, m0, Start,
                   End, reset, clock);
  output Load, Shift, Add_Shift, Sub, Enable, Ready;
  input m0, Start, End, clock, reset;
  reg state, next_state;
  parameter S0 = 0, S1 = 1;

  always @ (posedge clock or posedge reset)     // State transitions
      if (reset) state <= S0; else state <= next_state;

  always @ (state or Start or m0)  // Next state logic
     case (state)
        S0: if (Start) next_state = S1; else next_state = S0;
        S1: if (End) next_state = S0; else next_state = S1;
     endcase
  // control output logic
  assign Add_Shift = (state==S1) & m0;
  assign Sub = (state==S1) & End & m0;
  assign Shift = (state==S1) & ~m0;
  assign Load = (state==S0) && Start;
  assign Enable = (state==S1);
  assign Ready = (state==S0) && ~reset;
endmodule
```

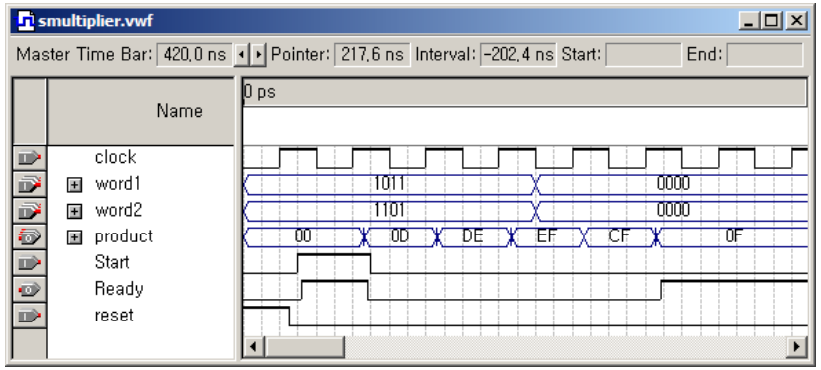

signed 곱셈기

```
module smultiplier(product, Ready,  word1, word2, Start, clock, reset);
  parameter N=4;
  output [2*N-1:0]  product;
  output Ready;
  input [N-1:0] word1, word2;
  input Start, clock, reset;
  wire  m0, Load, Shift, Add_Shift, Sub, Enable, End;
  wire [3:0] count;

  Datapath u1 (product, m0, word1, word2, Load, Shift, Add_Shift, Sub,
               reset, clock);
  Controller u2 (Load, Shift, Add_Shift, Sub, Enable, Ready, m0, Start,
                 End, reset, clock);
  counter u3 (count, Enable, Start & ~Enable, clock);
  assign End = (count==N-1);
endmodule
```

counter는 앞의 곱셈기와 같음

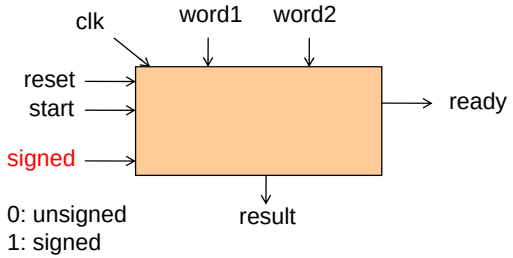
signed 곱셈기: Verilog description

- simulation
  - (-5)*(-3) = 15

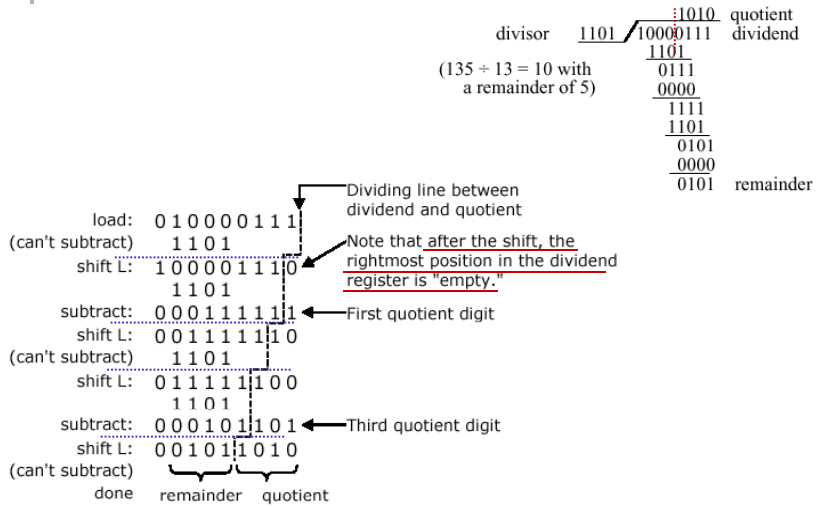


통합 곱셈기

- 통합 곱셈기
  - 제어신호에 따라서 unsigned 곱셈 또는 signed 곱셈 수행
  - 두 가지 곱셈이 모두 가능하도록 datapath와 controller를 수정

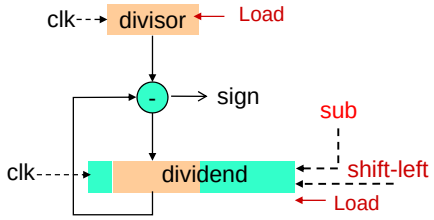


순차 나눗셈 - unsigned



순차 나눗셈기 – unsigned

- datapath
  - dividend와 remainder/quotient를 함께 사용
    - 나눗셈 후에 dividend는 remainder/quotient로 대치됨
  - divisor를 shift-right하는 것 대신에 dividend를 shift-left함
  - n+1 비트 subtractor 사용



나눗셈 알고리즘

Dividend (9-bit) ← word1 (8-bit), Divisor ← word1 (4-bit)
repeat 4 times **do**
- Shift Left Dividend
- Diff = Dividend[8:4] – {0,Divisor[3:0]} // 5-bit subtraction

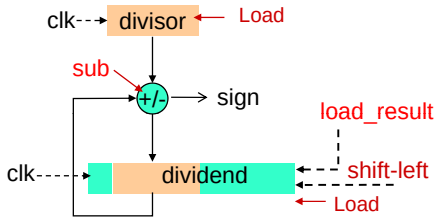
  if (Diff < 0) // Diff[4]==1
  Dividend[0] ← 0 // Quotient = 0
  else // Diff[4]==0
  Dividend[0] ← 1, Dividend[8:4] ← Diff // Quotient = 1

end
Remainder ← Dividend[7:4], Quotient← Dividend[3:0]

if (Dividend[8:4]>=Divisor) then Overflow

순차 나눗셈기 - signed

- datapath
  - n+1 비트 adder/subtractor 사용



나눗셈 알고리즘

Dividend (8-bit) ← word1 (8-bit), Divisor ← word1 (4-bit)

repeat 4 times **do**

- Shift Left Dividend
- if (Dividend and Divisor have same sign)
  - Result = Dividend[8:4] – {Divisor[3],Divisor[3:0]}

  else
  - Result = Dividend[8:4] + {Divisor[3],Divisor[3:0]}
- if (Dividend and Result have same sign or Result is 0)
  - Dividend[0] ← 1, Dividend[8:4] ← Result // Quotient = 1

  else
  - Dividend[0] ← 0 // Quotient = 0

end

Remainder ← Dividend[7:4]

if (word1 and word2 have same sign) Quotient← Dividend[3:0]

else Quotient ← 2's complement of Dividend[3:0]