13주차 결과보고서

전공: 아트&테크놀로지학과 학년: 4학년 학번: 20191048 이름: 김도솔

**1.**

**4-bit Shift Register**

* Verilog source code

|  |  |
| --- | --- |
| Design Source Code | Simulation Source Code |
| `timescale 1ns / 1ps  module sr(clk, rst, in, out);  input clk, rst, in;  output[3:0] out;  reg[3:0] out;    initial out = 4'b0000;    always @(posedge clk) begin  if(rst == 1'b1)  out <= 4'b0000;  else begin  out = out >> 1;  out[3] = in;  end  end    endmodule | `timescale 1ns / 1ps  module sr\_tb;  reg clk, rst, in;  wire[3:0] out;  sr u\_sr (  .clk (clk ),  .rst (rst ),  .in (in ),  .out (out )  );  initial begin  clk = 1'b0;  rst = 1'b0;  in = 1'b1;  end  always #10 clk = ~clk;  always #50 in = ~in;  always #200 rst = ~rst;  initial begin  #1000  $finish;  end  endmodule |

위는 Verilog 코드로 4-bit Shift Register를 구현한 것이다. sr 모듈은 3개의 입력(clk, rst, in)과 1개의 출력(out)을 가지고 있다. 이때 clk는 클럭 신호, rst는 리셋 신호, in은 1 bit 입력, out은 4 bit 출력을 의미한다. 클럭의 상승 에지에서 리셋 신호가 활성화되면(rst가 1이면) out은 0000으로 초기화된다. 그렇지 않으면 out을 오른쪽으로 1 bit 만큼 시프트하고, 가장 왼쪽 비트(out[3])에 입력신호 in을 넣는다. 이렇게 하면 shift register는 클럭의 상승 에지마다 저장된 데이터를 한 칸씩 이동시키고 가장 왼쪽 비트를 in 값으로 업데이트할 수 있다.

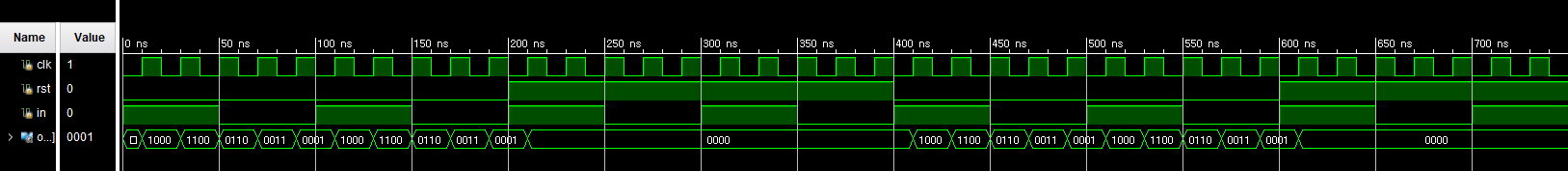
* Schematic Diagram

도표, 텍스트, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드로 Schematic Diagram을 생성한 결과이다. 이를 통해 작성한 코드가 회로에 어떤 식으로 구현되는지 알 수 있다.

* Simulation



위에서 작성한 Verilog 코드의 시뮬레이션 결과이다. 시간에 따라 세 입력 clk, rst, in의 값을 바꾸면 그에 따라 4 bit 출력 out의 값이 변화하는 것을 확인할 수 있다. 구현한 4-bit Shift Register가 각 입력 값에 따라 어떻게 동작하는지 아래 출력 예시들을 통해 좀 더 자세히 살펴보겠다.

* 출력 예시

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk와 rst의 초기 값은 0이고 in의 초기 값은 1이다. out도 design source code에서 설정한 초기 값 0000을 출력한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk는 1, rst는 0, in은 1의 값을 가진다. 이때 레지스터는 out의 각 비트를 오른쪽으로 한 칸씩 shift하고 출력의 가장 왼쪽 비트에 in의 값을 할당한다. 따라서 출력 값은 1000이다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 0이므로 rst, in의 값에 상관 없이 이전 출력 값을 유지한다. 카운터는 클럭의 상승 에지에만 상태 값을 업데이트하기 때문에 이와 같은 현상이 반복되므로 뒤의 출력 예시에서 clk가 0인 경우를 생략하겠다.

스크린샷, 회로이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0, in은 1의 값을 가진다. 레지스터는 out의 각 비트를 오른쪽으로 한 칸씩 shift하고 출력의 가장 왼쪽 비트에 in의 값을 할당한다. 따라서 출력 값은 1100이다.

스크린샷, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0, in은 0의 값을 가진다. 레지스터는 out의 각 비트를 오른쪽으로 한 칸씩 shift하고 출력의 가장 왼쪽 비트에 in의 값을 할당한다. 따라서 출력 값은 0110이다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0, in은 0의 값을 가진다. 레지스터는 out의 각 비트를 오른쪽으로 한 칸씩 shift하고 출력의 가장 왼쪽 비트에 in의 값을 할당한다. 따라서 출력 값은 0011이다.

스크린샷, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0, in은 0의 값을 가진다. 레지스터는 out의 각 비트를 오른쪽으로 한 칸씩 shift하고 출력의 가장 왼쪽 비트에 in의 값을 할당한다. 따라서 출력 값은 0001이다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0, in은 1의 값을 가진다. 레지스터는 out의 각 비트를 오른쪽으로 한 칸씩 shift하고 출력의 가장 왼쪽 비트에 in의 값을 할당한다. 따라서 출력 값은 1000이다. rst 신호가 활성화 되기 전까지 계속해서 같은 패턴이 나타난다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 0, rst는 1, in은 1의 값을 가진다. 리셋 신호가 활성화 되었지만 레지스터는 클럭의 상승 에지에만 값을 업데이트하기 때문에 이전 출력 값을 유지한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 1, in은 1의 값을 가진다. 리셋 신호가 활성화 되어 레지스터는 0000으로 초기화된다. 리셋 신호가 활성화된 동안 레지스터는 계속해서 출력 값을 0000으로 유지한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> rst가 0으로 리셋 신호가 비활성화 되었지만 clk가 0이므로 이전 출력 값인 0000을 유지한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0, in은 1의 값을 가진다. 레지스터는 out의 각 비트를 오른쪽으로 한 칸씩 shift하고 출력의 가장 왼쪽 비트에 in의 값을 할당한다. 따라서 출력 값은 1000이 된다. 이후로도 리셋 신호가 활성화되기 전까지 클럭의 상승 에지마다 같은 동작을 반복한다.

**2.**

**4-bit Ring counter**

* Verilog source code

|  |  |
| --- | --- |
| Design Source Code | Simulation Source Code |
| `timescale 1ns / 1ps  module rc(  clk, rst, out  );    input clk, rst;  output[3:0] out;  reg[3:0] out;    initial out = 4'b1000;    always @(posedge clk) begin  if(rst == 1'b1)  out = 4'b1000;  else begin  out[0] <= out[1];  out[1] <= out[2];  out[2] <= out[3];  out[3] <= out[0];  end  end    endmodule | `timescale 1ns / 1ps  module rc\_tb;  reg clk, rst;  wire[3:0] out;  rc u\_rc (  .clk (clk ),  .rst (rst ),  .out (out )  );  initial begin  clk = 1'b0;  rst = 1'b0;  end  always #10 clk = ~clk;  always #200 rst = ~rst;  initial begin  #1000  $finish;  end  endmodule |

위는 Verilog 코드로 4-bit Ring counter를 구현한 것이다. rc 모듈은 2개의 입력(clk, rst)과 1개의 출력(out)을 가지고 있다. 이때 clk는 클럭 신호, rst는 리셋 신호를 나타내고 out은 4 bit 출력을 의미한다. 클럭의 상승 에지에서 리셋 신호가 활성화되면(rst가 1이면) out은 1000으로 초기화된다. 그렇지 않으면 out의 각 비트가 그 다음 비트로 이동한다. 즉 out[0]은 out[1]로, out[1]은 out[2]로, out[2]는 out[3]로, 그리고 out[3]는 out[0]으로 이동한다. 이렇게 하면 클럭의 상승 에지마다 비트가 한 위치에서 다음 위치로 계속해서 이동하는 Ring counter의 순환 구조를 구현할 수 있다.

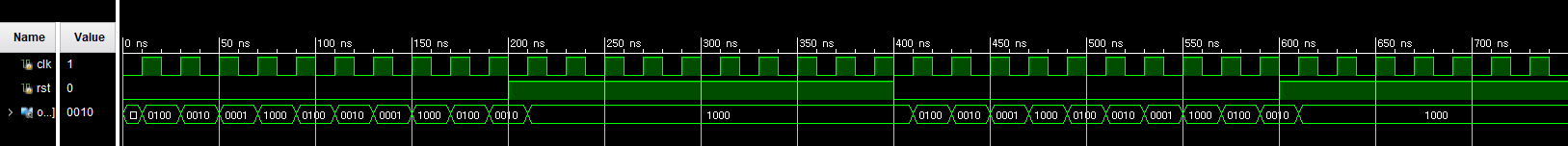
* Schematic Diagram

도표, 텍스트, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드로 Schematic Diagram을 생성한 결과이다. 이를 통해 작성한 코드가 회로에 어떤 식으로 구현되는지 알 수 있다.

* Simulation



위에서 작성한 Verilog 코드의 시뮬레이션 결과이다. 시간에 따라 두 입력 clk, rst의 값을 바꾸면 그에 따라 4 bit 출력 out의 값이 변화하는 것을 확인할 수 있다. 구현한 4-bit Ring counter가 각 입력 값에 따라 어떻게 동작하는지 아래 출력 예시들을 통해 좀 더 자세히 살펴보겠다.

* 출력 예시

스크린샷, 폰트, 라인, 텍스트이(가) 표시된 사진

자동 생성된 설명

> clk와 rst의 초기 값은 둘다 0이다. out도 design source code에서 설정한 초기 값 1000을 출력한다.

스크린샷, 텍스트, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

> clk는 1이고, rst는 0이다. 따라서 링 카운터는 out의 각 비트를 그 다음 비트로 이동시킨다. 이때 마지막 비트 값은 첫 번째 비트에 할당된다. 따라서 출력 값은 0100이다.

스크린샷, 라인, 텍스트, 폰트이(가) 표시된 사진

자동 생성된 설명

> clk가 0이므로 링 카운터는 이전 출력 값을 유지한다. 링 카운터는 클럭의 상승 에지에만 상태 값을 업데이트하기 때문에 이와 같은 현상이 계속 반복되므로 뒤의 출력 예시에서 clk가 0인 경우를 생략하겠다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0이다. 링 카운터는 out의 각 비트를 그 다음 비트로 이동시킨다. 따라서 출력 값은 0010이다.

스크린샷, 텍스트, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0이다. 링 카운터는 out의 각 비트를 그 다음 비트로 이동시킨다. 따라서 출력 값은 0001이다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0이다. 링 카운터는 out의 각 비트를 그 다음 비트로 이동시킨다. 마지막 비트 값인 1이 첫 번째 비트에 할당되어 출력 값은 1000이 된다. 이는 초기 값과 동일하다. 링 카운터는 rst 신호가 활성화 되기 전까지 계속해서 같은 패턴(1000-0100-0010-0001)을 반복한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 0, rst는 1의 값을 가진다. 리셋 신호가 활성화 되었지만 링 카운터는 클럭의 상승 에지에만 값을 업데이트하기 때문에 이전 출력 값을 유지한다.

스크린샷, 라인, 텍스트, 폰트이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 1, in은 1의 값을 가진다. 리셋 신호가 활성화 되어 링 카운터는 1000으로 초기화된다. 리셋 신호가 활성화된 동안 링 카운터는 계속해서 출력 값을 1000으로 유지한다.

스크린샷, 텍스트, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

> rst가 0으로 리셋 신호가 비활성화 되었지만 clk가 0이므로 이전 출력 값인 1000을 유지한다.

스크린샷, 텍스트, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst는 0의 값을 가진다. 링 카운터는 out의 각 비트를 그 다음 비트로 이동시킨다. 따라서 출력 값은 0010이 된다. 이후로도 리셋 신호가 활성화되기 전까지 클럭의 상승 에지마다 동일한 패턴을 반복한다.

**3.**

**4-bit Up/Down counter**

* Verilog source code

|  |  |
| --- | --- |
| Design Source Code | Simulation Source Code |
| `timescale 1ns / 1ps  module udc(  clk, rst, state, out, seg, digit  );    input clk, rst, state;  output[3:0] out;  output[6:0] seg;  output digit;  reg[3:0] out;  reg[6:0] seg;  reg digit;    initial out <= 4'b0000;  initial digit = 1;    always @(posedge clk) begin  if (rst == 1'b1)  out <= 4'b0000;  else begin  if(state == 1'b1) begin  if (out == 4'b1111) begin  out <= 4'b0000;  end  else begin  out <= out + 4'b0001;  end  seg <= 7'b0111110;  end  else begin  if (out == 4'b0000) begin  out <= 4'b1111;  end  else begin  out <= out - 4'b0001;  end  seg <= 7'b0111101;  end  end  end  endmodule | `timescale 1ns / 1ps  module udc\_tb;  reg clk, rst, state;  wire[3:0] out;  wire[6:0] seg;  wire digit;  udc u\_udc (  .clk (clk ),  .rst (rst ),  .state (state ),  .out (out ),  .seg (seg ),  .digit (digit )  );  initial begin  clk = 1'b0;  rst = 1'b0;  state = 1'b1;  end  always #10 clk = ~clk;  always #400 state = ~state;  always #600 rst = ~rst;  initial begin  #1000  $finish;  end  endmodule |

위는 Verilog 코드로 4-bit Up/Down counter를 구현한 것이다. udc 모듈은 3개의 입력(clk, rst, state)과 3개의 출력(out, seg, digit)을 가지고 있다. 이때 clk는 클럭 신호, rst는 리셋 신호를 나타내고 state는 카운터의 동작 상태를 나타내는데, state가 1이면 Up counter로 동작하고, 0이면 Down counter로 동작한다. out은 counter의 4 bit 출력, seg와 digit은 7 세그먼트 디스플레이의 출력을 위한 변수이다. Up counter로 동작하는 경우에는 7 세그먼트 디스플레이에 U를 출력하고, Down counter로 동작하는 경우에는 7 세그먼트 디스플레이에 d를 출력하도록 설계했다. 이 모듈은 클럭의 상승 에지 마다 동작하며, 리셋이 활성화되면 출력 out을 0000으로 초기화한다. 그렇지 않으면 state 값에 따라 out 값을 증가 또는 감소시킨다. Up counter로 동작 시, 출력 값의 최댓값(1111)에 도달하면 다음 출력 값으로 0000을 할당하고, Down counter로 동작 시 출력의 최솟값(0000)에 도달하면 다음 출력 값으로 1111을 할당하도록 구성했다.

* Schematic Diagram

도표, 라인, 텍스트, 평면도이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드로 Schematic Diagram을 생성한 결과이다. 이를 통해 작성한 코드가 회로에 어떤 식으로 구현되는지 알 수 있다.

* Simulation

스크린샷, 라인, 다채로움이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드의 시뮬레이션 결과이다. 시간에 따라 세 입력 clk, rst, state의 값을 바꾸면 그에 따라 두 출력 out과 seg의 값이 변화하는 것을 확인할 수 있다. 출력 digit은 입력 값에 상관없이 항상 1이다. 구현한 4-bit Up/Down counter가 각 입력 값에 따라 어떻게 동작하는지 아래 출력 예시들을 통해 좀 더 자세히 살펴보겠다.

* 출력 예시

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 0, rst가 0, state가 1이다. 이는 Simulation Source Code에서 설정한 각 입력의 초기 값이다. out 또한 Design Source Code에서 설정한 초기 값 0000을 출력한다. seg는 초기 값을 설정하지 않아 X로 표시된다. digit은 입력 값에 관계 없이 항상 1로 설정해두었기에 설명을 생략하겠다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst가 0, state가 1이다. 따라서 이는 Up counter로 동작한다. Up counter는 클럭의 상승 에지마다 출력값을 1씩 증가시키므로 out은 0001을 출력한다. seg는 Up counter로 동작하는 동안 7 세그먼트 디스플레이에 U를 출력하기 위해 계속해서 0111110의 값을 가진다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 0이므로 카운터는 이전 출력 값을 유지한다. 이 카운터는 클럭의 상승 에지마다 상태 값을 업데이트하기 때문에 이와 같은 현상이 계속 반복되므로 뒤의 출력 예시에서 clk가 0인 경우를 생략하겠다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst가 0, state가 1이다. 따라서 이는 계속해서 Up counter로 동작한다. 따라서 out은 1 증가된 0010을 출력하고, seg는 0111110의 값을 가진다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> state가 1인 동안 카운터는 계속해서 Up counter로 동작하므로 클럭의 상승 에지마다 out의 값을 1씩 증가시킨다. 위는 clk가 1, rst가 0, state가 1인 경우이다. 이때 out은 최댓값인 1111을 출력한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst가 0, state가 1이다. 위 예시에서 최댓값인 1111을 넘어가면 다시 0000을 출력하는 것을 알 수 있다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 0, rst가 0, state가 0이다. state가 0이 되었지만 카운터는 클럭의 상승 에지에만 값을 업데이트하기 때문에 이전 출력 값을 유지한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst가 0, state가 0이다. 따라서 이는 Down counter로 동작한다. Down counter는 클럭의 상승 에지마다 출력값을 1씩 감소시키므로 out은 이전 출력 값 0100에서 1 감소한 0011을 출력한다. seg는 Down counter로 동작하는 동안 7 세그먼트 디스플레이에 d를 출력하기 위해 0111101의 값을 가진다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> state가 0인 동안 카운터는 계속해서 Down counter로 동작하므로 클럭의 상승 에지마다 out의 값을 1씩 감소시킨다. 위는 clk가 1, rst가 0, state가 0인 경우이다. 이때 out은 최솟값인 1111을 출력한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst가 0, state가 0이다. 위 예시에서 최솟값인 0000 보다 작아지면 1111을 출력하는 것을 알 수 있다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 0, rst가 1, state가 0이다. 리셋 신호가 활성화 되었지만 카운터는 클럭의 상승 에지에만 값을 업데이트하기 때문에 이전 출력 값 1010을 유지한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst가 1, state가 0이다. 리셋 신호가 활성화 되어 카운터는 0000으로 초기화된다. 리셋 신호가 활성화된 동안 카운터는 계속해서 출력 값을 0000으로 유지한다.

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

> clk가 1, rst가 1, state가 1이다. 리셋 신호가 활성화된 동안 state 값이 변화해도 카운터는 계속해서 출력 값을 0000으로 유지하는 것을 확인할 수 있다.

**4.**

위의 세 모듈 모두 시뮬레이션과 FPGA 보드 상에서 아래와 같이 의도한 대로 동작하는 것을 확인했다. 따라서 각 모듈이 올바르게 구현되었음을 확인할 수 있었다.

* 4-bit Shift Register: 클럭의 상승 에지마다 저장된 데이터를 한 칸씩 이동시키고 가장 왼쪽 비트를 in 값으로 업데이트하는 것을 관찰할 수 있었다.
* 4-bit Ring Counter: 클럭의 상승 에지마다 비트가 한 위치에서 다음 위치로 계속해서 이동하는 순환구조를 확인할 수 있었다.
* 4-bit Up/Down Counter: 클럭의 상승 에지 마다 동작하며, state 값에 따라 Up counter 혹은Down counter로 동작하는 것을 확인했다. 특히 Up counter로 동작 시, 출력 값의 최댓값(1111)에 도달하면 다음 출력 값으로 0000을 할당하고, Down counter로 동작 시 출력의 최솟값(0000)에 도달하면 다음 출력 값으로 1111을 할당하는 것이 관찰되었다.

**5.**

이번 실습에서 구현한 4-bit Ring Counter의 verilog 코드를 변형하여 4-bit Johnson Counter를 구현할 수 있다.

|  |  |
| --- | --- |
| Design Source Code | Simulation Source Code |
| `timescale 1ns / 1ps  module jc(  clk, rst, out  );    input clk, rst;  output[3:0] out;  reg[3:0] out;    initial out = 4'b1000;    always @(posedge clk) begin  if(rst == 1'b1)  out = 4'b1000;  else begin  out[0] <= out[1];  out[1] <= out[2];  out[2] <= out[3];  **out[3] <= ~out[0];**  end  end    endmodule | `timescale 1ns / 1ps  module jc\_tb;  reg clk, rst;  wire[3:0] out;  jc u\_jc (  .clk (clk ),  .rst (rst ),  .out (out )  );  initial begin  clk = 1'b0;  rst = 1'b0;  end  always #10 clk = ~clk;  always #200 rst = ~rst;  initial begin  #1000  $finish;  end  endmodule |

jc 모듈은 2개의 입력(clk, rst)과 1개의 출력(out)을 가지고 있다. 이때 clk는 클럭 신호, rst는 리셋 신호를 나타내고 out은 4 bit 출력을 의미한다. 클럭의 상승 에지에서 리셋 신호가 활성화되면(rst가 1이면) out은 1000으로 초기화된다. 그렇지 않으면 out의 세 비트가 그 다음 비트로 이동한다. 즉 out[0]은 out[1]로, out[1]은 out[2]로, out[2]는 out[3]로 이동한다. 가장 왼쪽 비트인 out[3]에는 out[0]를 NOT 연산한 결과를 할당한다. 이로써 특정한 패턴의 존슨 카운터 동작을 구현할 수 있다.

스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명