

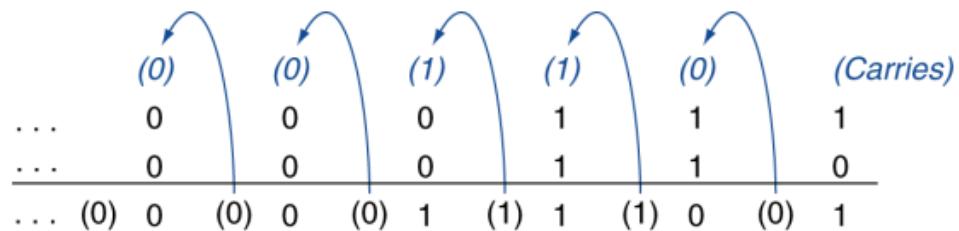


Chapter03_1.Arithmetic for Computers

Binary Addition

- Adding 6 and 7

$$\begin{array}{r} \dots 0000\ 0000\ 0000\ 0111 \\ + \dots 0000\ 0000\ 0000\ 0110 \\ \hline \dots 0000\ 0000\ 0000\ 1101 \end{array}$$



- Overflow in addition (덧셈에서 발생한 오버플로우)
 - Not possible when signs are different(기호가 다른 경우는 불가능하다.)
 - 결과의 크기는 피연산자보다 작아야만 한다.

Binary Subtraction

- Sub 6 from 7 : $7 - 6 = 7 + (-6)$

$$\begin{array}{r} \dots 0000\ 0000\ 0000\ 0111 \\ + \dots 1111\ 1111\ 1111\ 1010 \\ \hline \dots 0000\ 0000\ 0000\ 0001 \end{array}$$

2's complement representation of -6

- No overflow if signs of both operands are the same(두 연산자의 부호가 동일한 경우 오버플로우가 발생하지 않는다.)
- Overflow possible if operands have different signs(피연산자의 부호가 다른 경우 오버플로우가 가능하다.)

Overflow Detection

- Detecting overflow for 2's complement numbers(2의 보수가 적용된 수에서 발생하나 overflow를 탐색해본다.)
 - ex) In addition, it is the overflow if sign bit becomes 1(또한 부호 비트가 1이 되면 오버플로우가 발생한다.)

| Operation | Operand A | Operand B | Result indicating overflow |
|-----------|-----------|-----------|----------------------------|
| $A + B$ | ≥ 0 | ≥ 0 | < 0 |
| $A + B$ | < 0 | < 0 | ≥ 0 |
| $A - B$ | ≥ 0 | < 0 | < 0 |
| $A - B$ | < 0 | ≥ 0 | ≥ 0 |

- 위의 4가지 경우 모두 다 오버플로우가 발생한 경우이다.
- Overflow of Unsigned Integers(부호 없는 정수의 오버플로우)
 - Addition overflows if the sum is less than either of the addends(합계가 가수 중 하나보다 작으면 더하기가 오버플로우가 된다.)
 - Overflow if $A+B=S$ and $S < A$ or $S < B$ (S가 A, B보다 적어도 커야만 하는데, 이 둘 보다 작은 경우는 오버플로우가 발생한 경우다.)

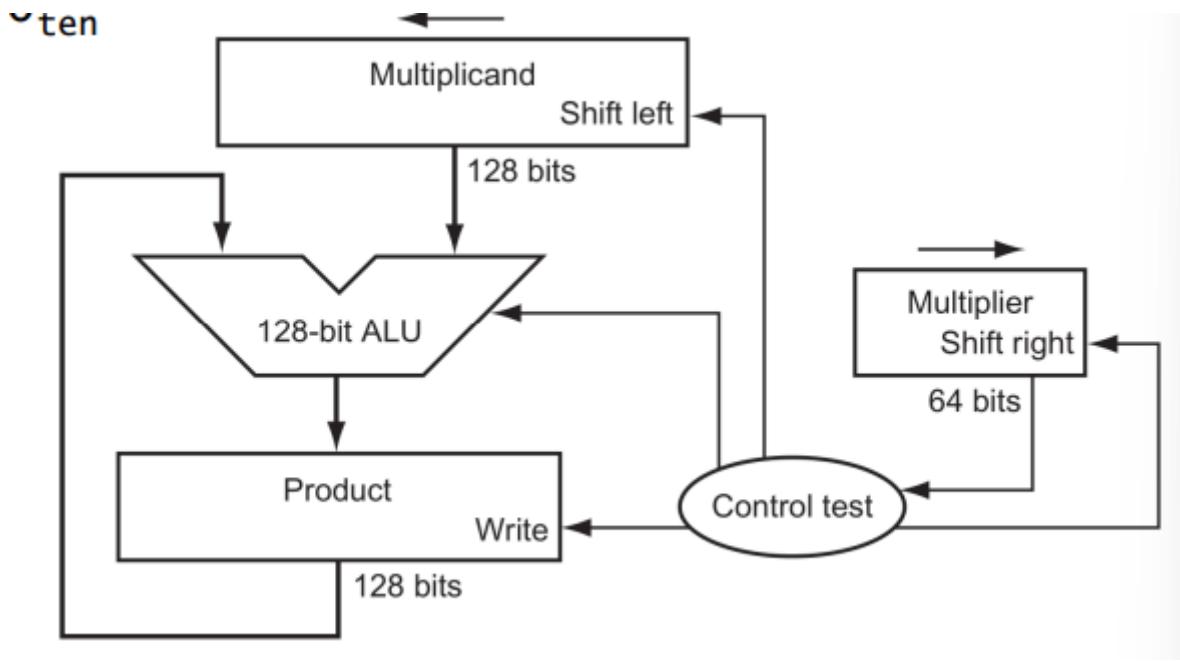
- How to handle overflow differs from language to language(각각의 언어에서 overflow를 다루는 방법)
 - C, java : ignore integer overflow(C, java는 overflow를 무시한다.)
 - Ada, Fortran : overflow must be notified to program(에이다, 포트란은 경고를 준다.)

Multiplication

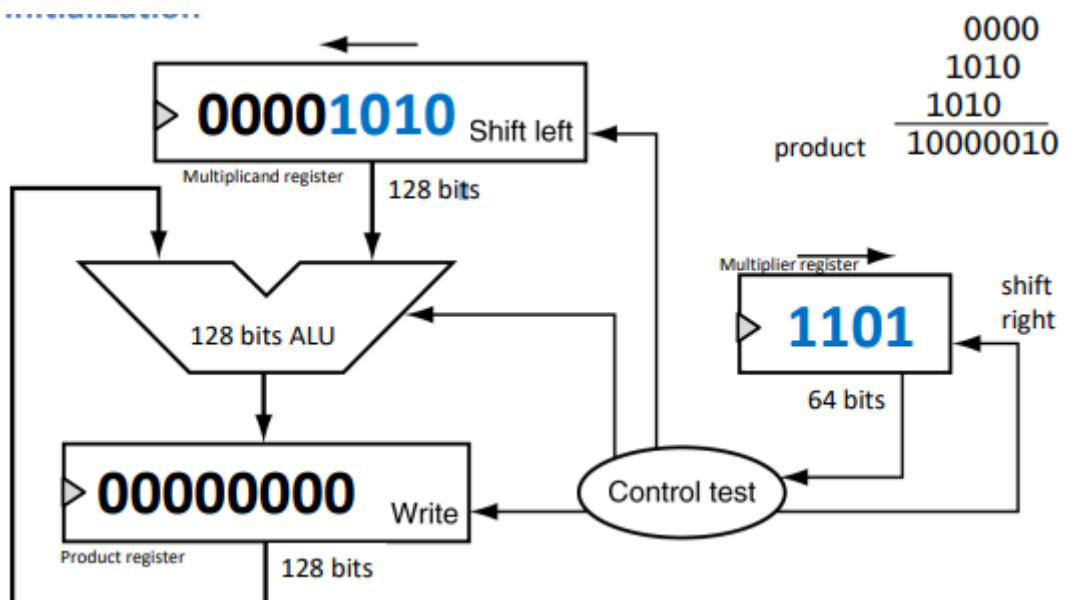
- steps of multiplication

Max Length of product =
length of multiplicand
+ length of multiplier

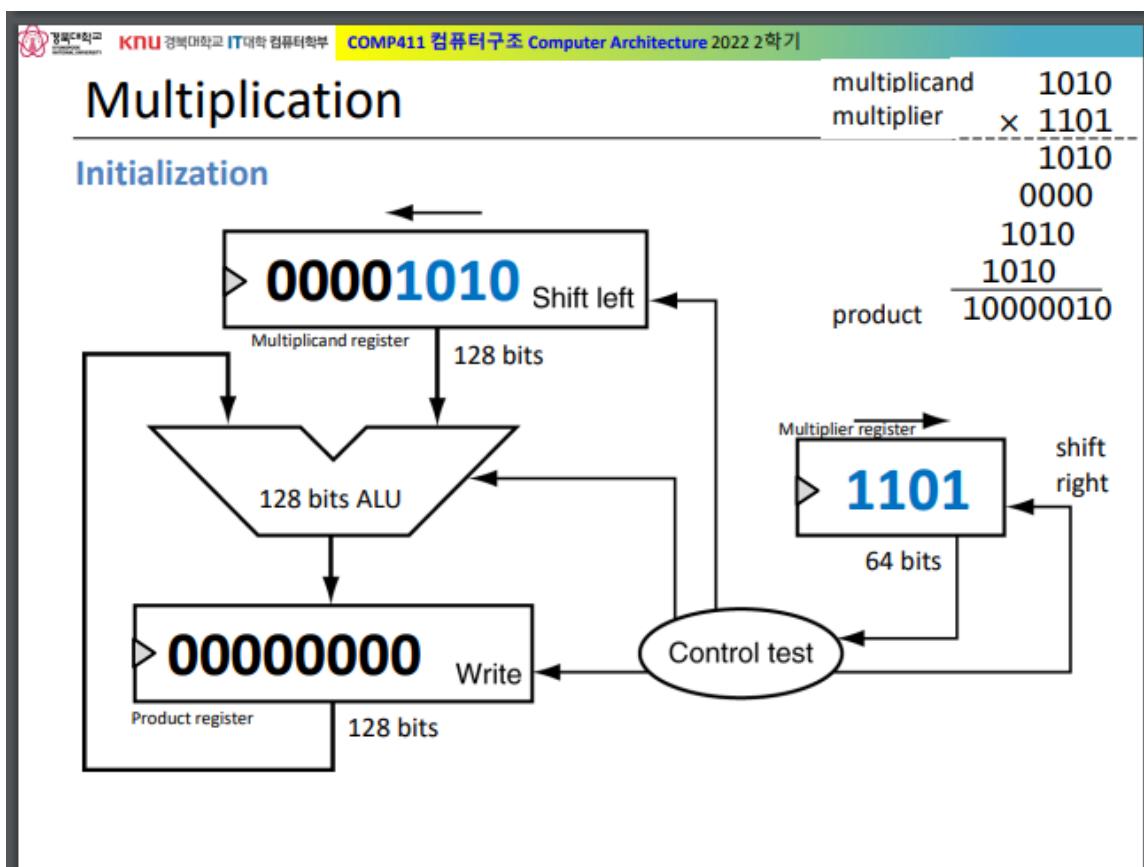
$$\begin{array}{r}
 \text{multiplicand} & 1000_{\text{ten}} \\
 \text{multiplier} & \times 1001_{\text{ten}} \\
 \hline
 & 1000 \\
 & 0000 \\
 & 0000 \\
 & 1000 \\
 \hline
 \text{product} & 1001000_{\text{ten}}
 \end{array}$$



- Multiplication Hardware(곱하기 작용에서 하드웨어 구조)
 - First version(우선 첫번째 버전부터 살펴 볼 것이다.)
 - 참고로 한 사이클 계산하는데, 레지스터 (product, Multiplier, Multiplicand)가 3개 니깐 CLK가 총 3번 나타나게 된다.
 - 성능향상을 위해 ALU의 비트수가 작은 것을 쓰는게 계산 속도가 더 빨라진다.
 - Sequential processing (순차적 프로그램)
- Multiplicand register(128bit)의 경우는 Shift left 를 수행하고, Multiplier register(64bit) 의 경우는 Shift right를 수행한다.



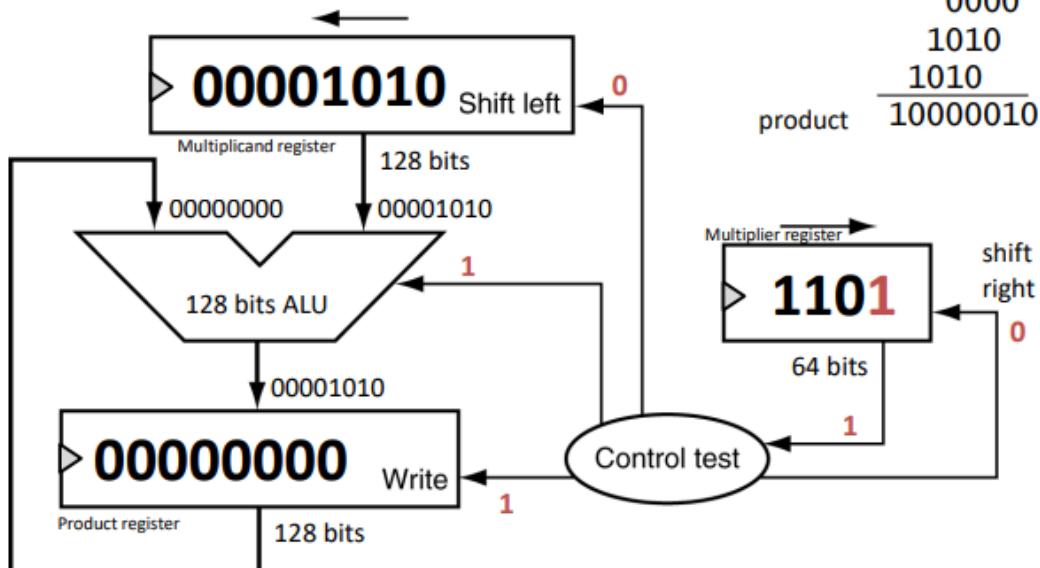
- 우선 연산자와 피연산자 자리에 숫자를 채워준다.
- ALU는 계산을 해 주는 하드웨어인데, 이 ALU는 값이 있던 없던 늘 계산을 수행중이다. 다만, CLK가 켜지고 control test가 신호를 줘야만 Write 자리에 숫자를 채워 줄 수 있다.
- 참고로 ALU는 늘 돌아가고 있기 때문에, 항상 1로 신호가 켜져있다.
- Control test는 Multiplier, Multiplicand, ALU, Write 모두 연결 돼 있다.
- 그럼 수행 동작을 순차적으로 살펴보도록 하자.
- 우선 처음 상태는 아래와 같다.



1. $ALU == 1, SII == 0, Srl == 0, Write == 1$

Multiplication

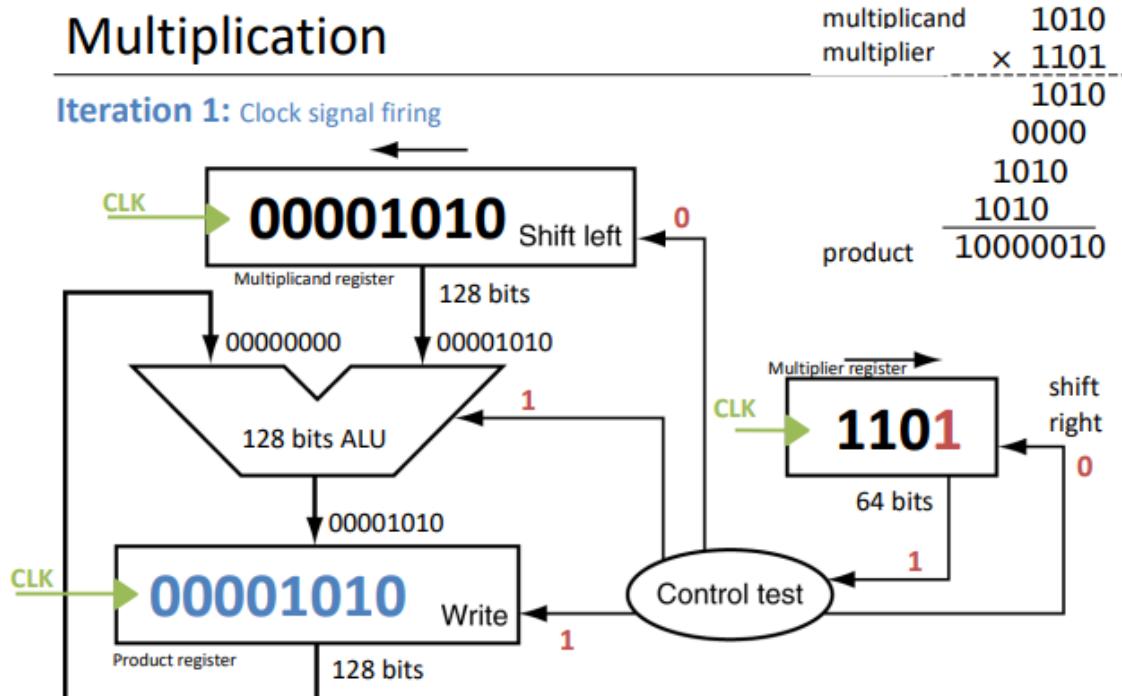
Iteration 1: Control signal set-up



1. CLK ON! ALU == 1, Sll == clk && 0, Srl == clk && 0, write == clk && 1

Multiplication

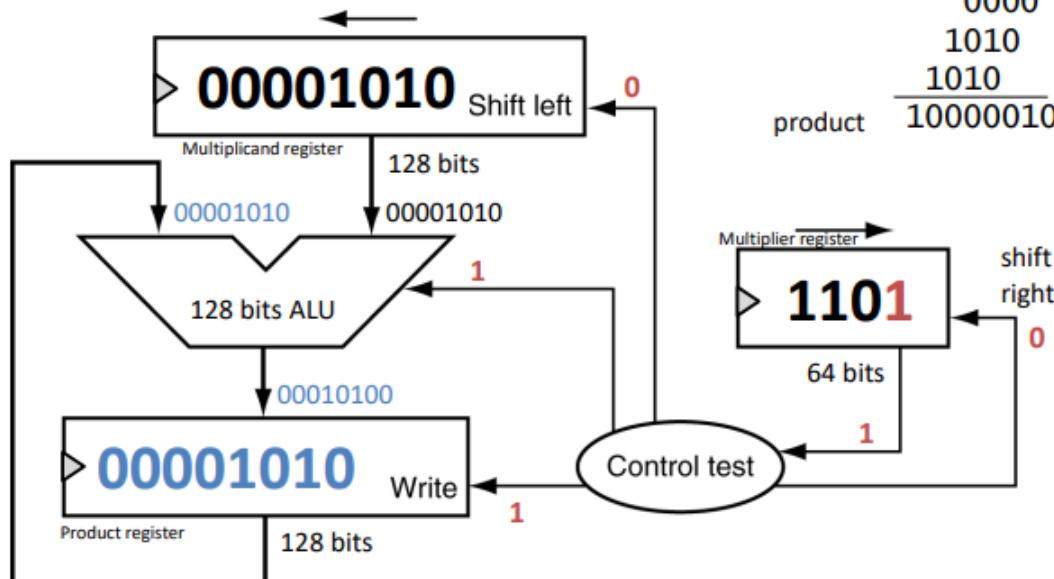
Iteration 1: Clock signal firing



1. CLK off! ALU == 1, Sll && Srl == 0, write == 1

Multiplication

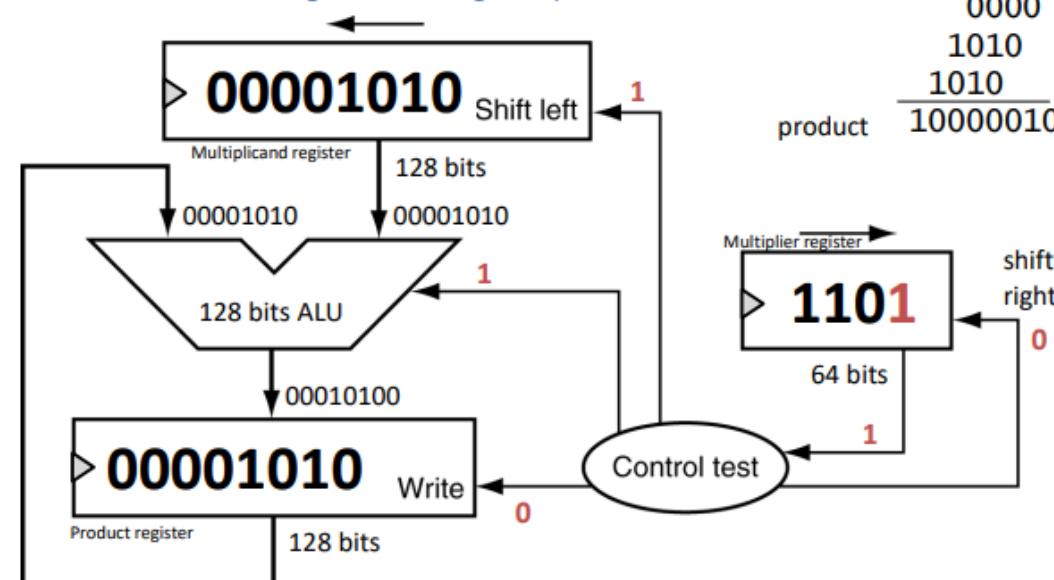
Iteration 1: Product register updated



1. SII == 1, ALU == 1, Write == 0, Srl == 0

Multiplication

Iteration 1: Control signal for shifting multiplicand

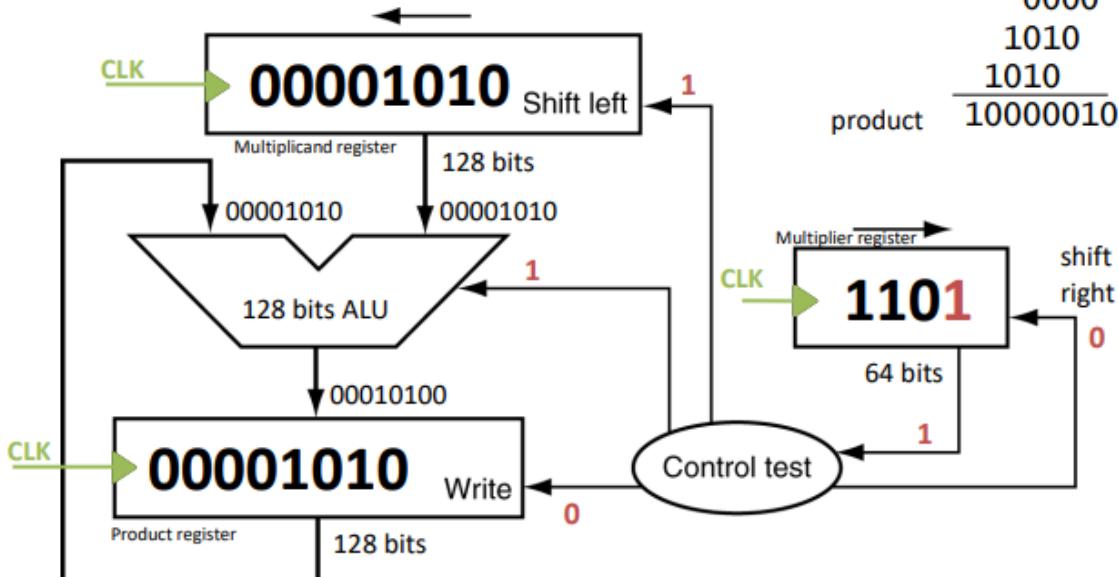


1. CLK On! ALU == 1, SII == CLK && 1, Srl == CLK && 0, write == CLK && 0

Multiplication

$$\begin{array}{r}
 \text{multiplicand} \quad 1010 \\
 \times \text{multiplier} \quad 1101 \\
 \hline
 1010 \\
 0000 \\
 1010 \\
 \hline
 1010 \\
 \hline
 10000010
 \end{array}$$

Iteration 1: Clock firing

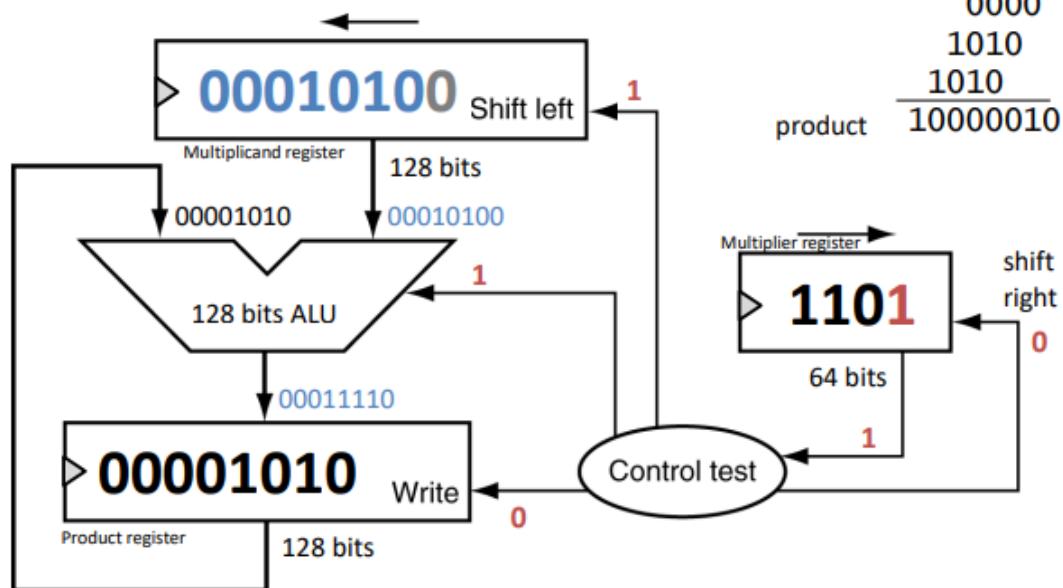


1. CLK off! SII → 수행 && 1, Srl == 0, ALU == 1, write == 0

Multiplication

$$\begin{array}{r}
 \text{multiplicand} \quad 1010 \\
 \times \text{multiplier} \quad 1101 \\
 \hline
 1010 \\
 0000 \\
 1010 \\
 \hline
 1010 \\
 \hline
 10000010
 \end{array}$$

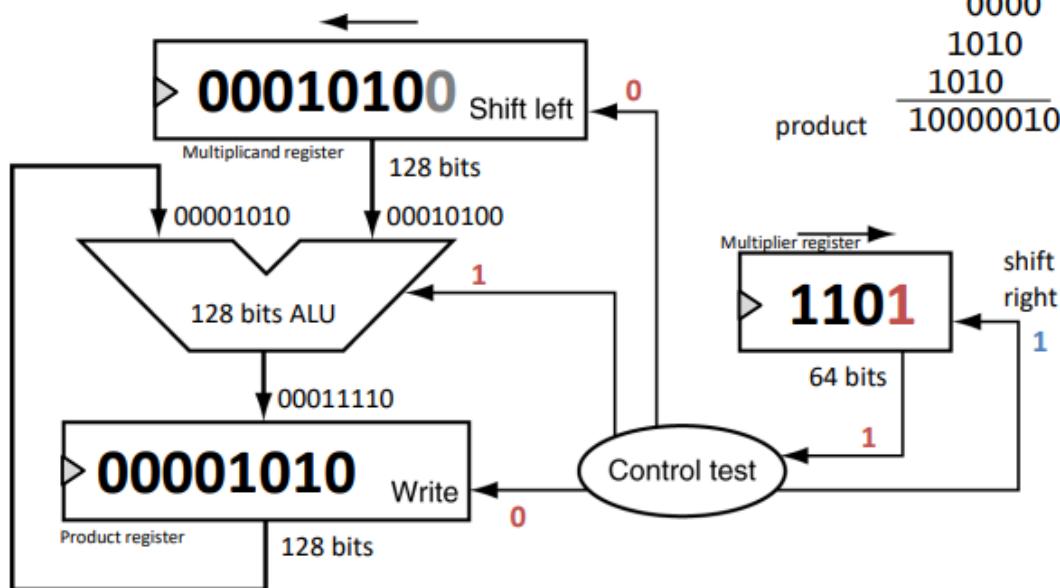
Iteration 1: Multiplicand shifted left



1. SII == 0, ALU ==1 , SRL == 1, write == 0

Multiplication

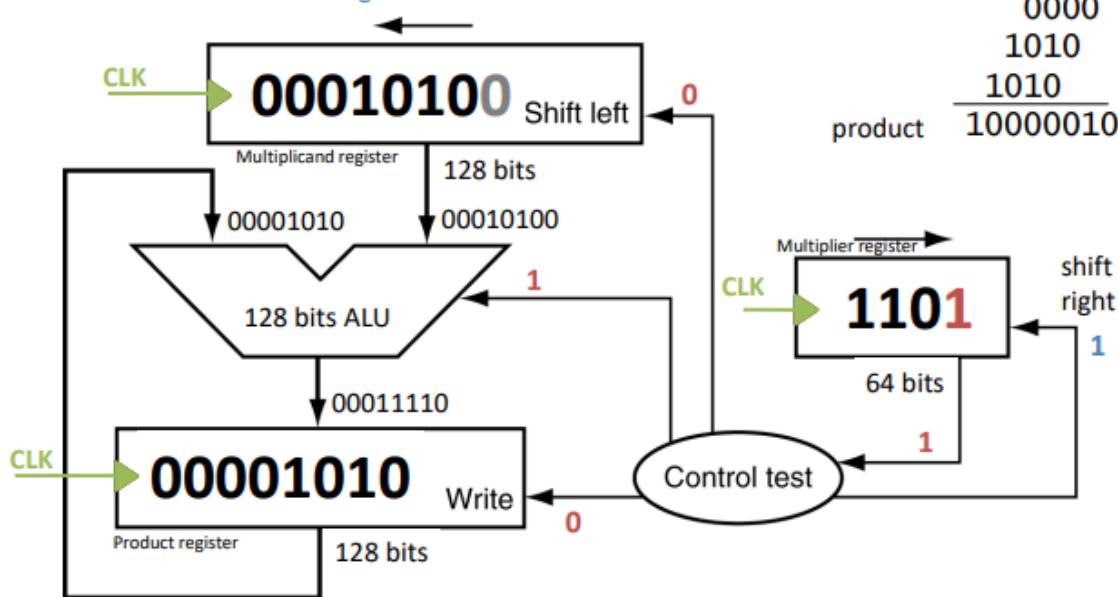
Iteration 1: Control signal for shifting multiplier



1. CLK On!!, sll == 0 && CLK, srl == 1 && CLK, write == 0, ALU == 1

Multiplication

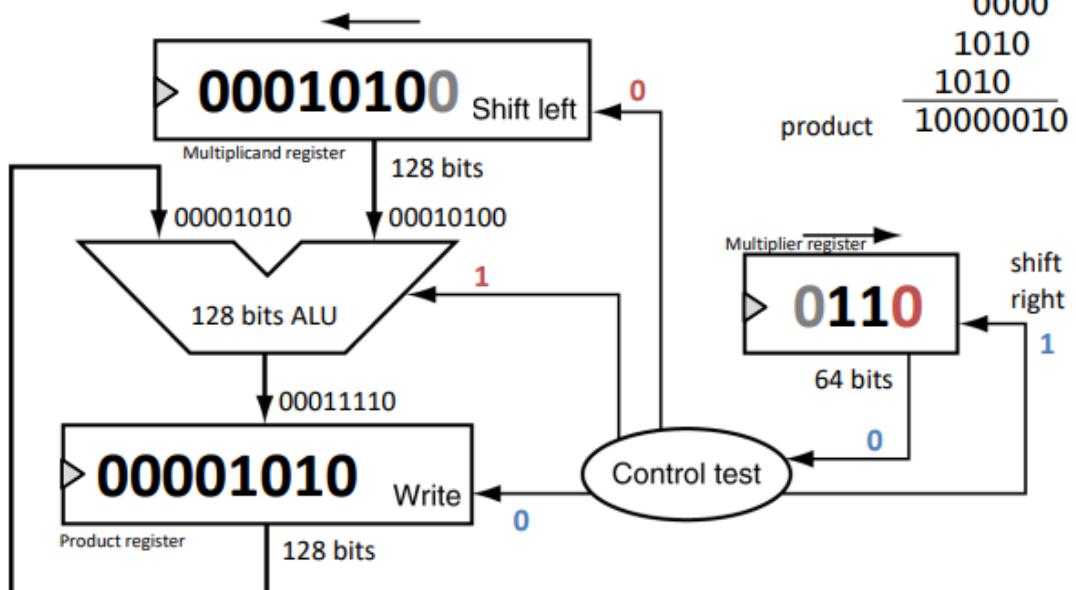
Iteration 1: Clock firing



1. CLK off!, Srl수행!!, ALU == 1, Sll == 0, write == 0

Multiplication

Iteration 1: Multiplier shifted

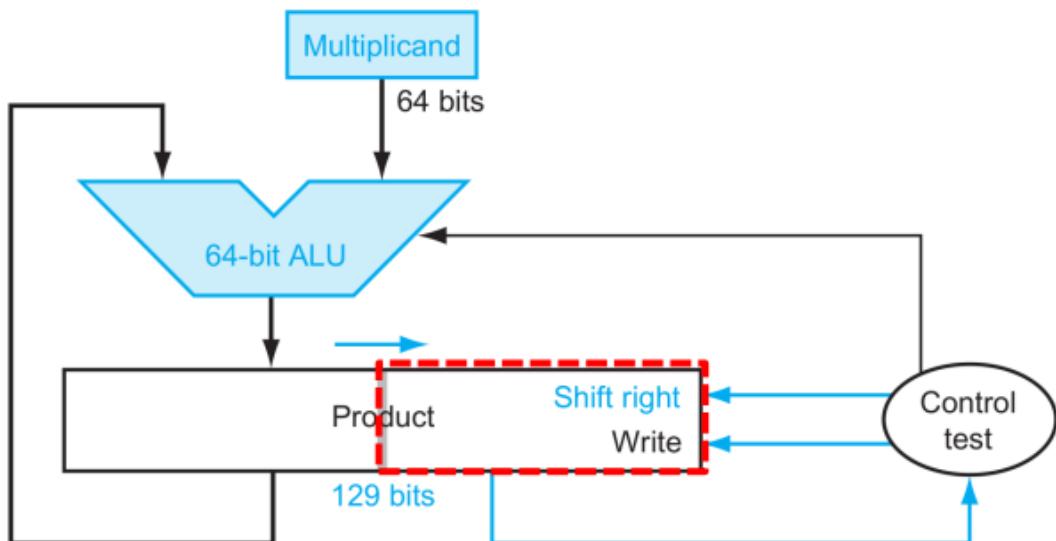


$$\begin{array}{r}
 \text{multiplicand} \quad 1010 \\
 \text{multiplier} \quad \times 1101 \\
 \hline
 & 1010 \\
 & 0000 \\
 & 1010 \\
 & \underline{1010} \\
 \text{product} & 10000010
 \end{array}$$

- 이상을 끝으로 이 회로의 첫 번째 단계는 끝이났다, 여기까지가 한 사이클이고 , 이제 3번만 더 반복한 후 , 최종적으로 write에 product값이 쓰여지면, 계산은 끝이 나게 된다.

Improving the Multiplication

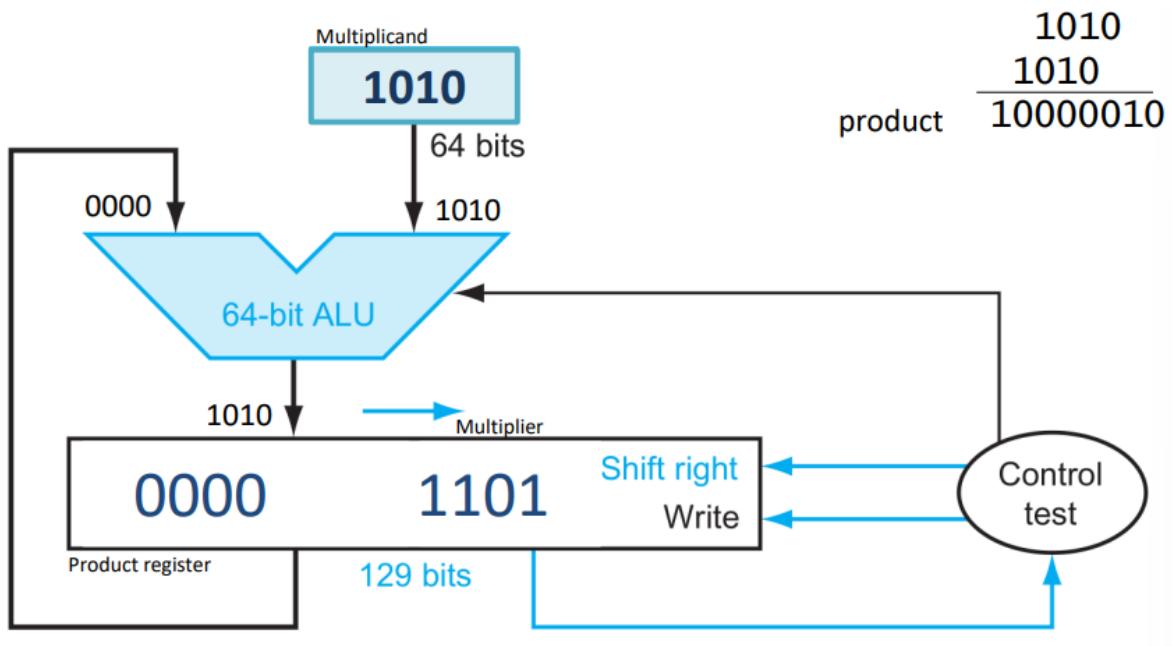
- 이번에는 위에서 보았던 더하기 논리 회로보다 좀 더 간결하고 효율성 높은 버전의 곱셈 회로를 살펴보고자 한다. 우선 기본 구성은 아래와 같다.



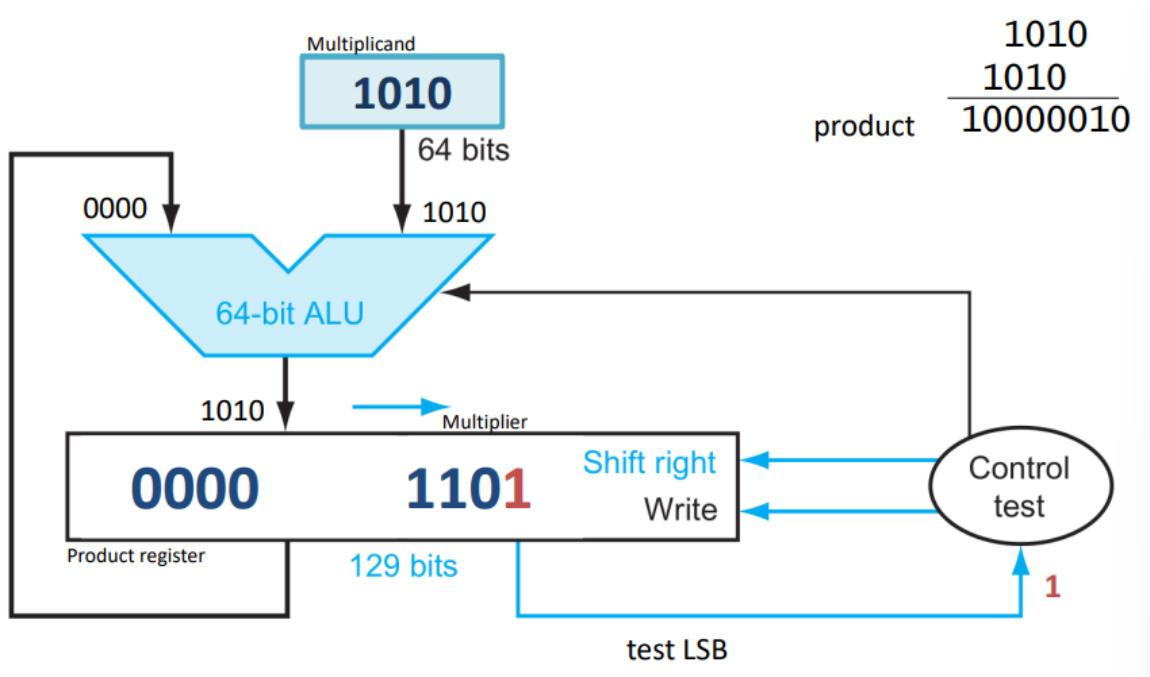
- 아까랑 비교해 보았을 때, 가장 큰 차이는 Multiplier 이 없어졌다. 대신 Product 가 129비트가 되었고, 그 중 오른쪽에 write가 쓰인다. 아마, write랑 Multiplier(64비트)이 반반씩 나누어 쓰나보다.
- 그런데 왜, 129비트인 걸까? 왜냐하면, addition과 shift를 병렬로 처리해서 한 사이클에 모든 연산이 가능하기 때문이다.
 - 한 사이클마다 Shift right 후 write를 한꺼번에 수행한다.
- 그리고 덧셈과 시프트가 동시에 일어난다. 아까와 같은 연산식을 수해하는 과정을 살펴보도록 하자.

$$\begin{array}{r}
 \text{multiplicand} \quad 1010 \\
 \text{multiplier} \quad \times \quad 1101 \\
 \hline
 & 1010 \\
 & 0000 \\
 & 1010 \\
 & \hline
 & 1010 \\
 \text{product} \quad \hline
 & 10000010
 \end{array}$$

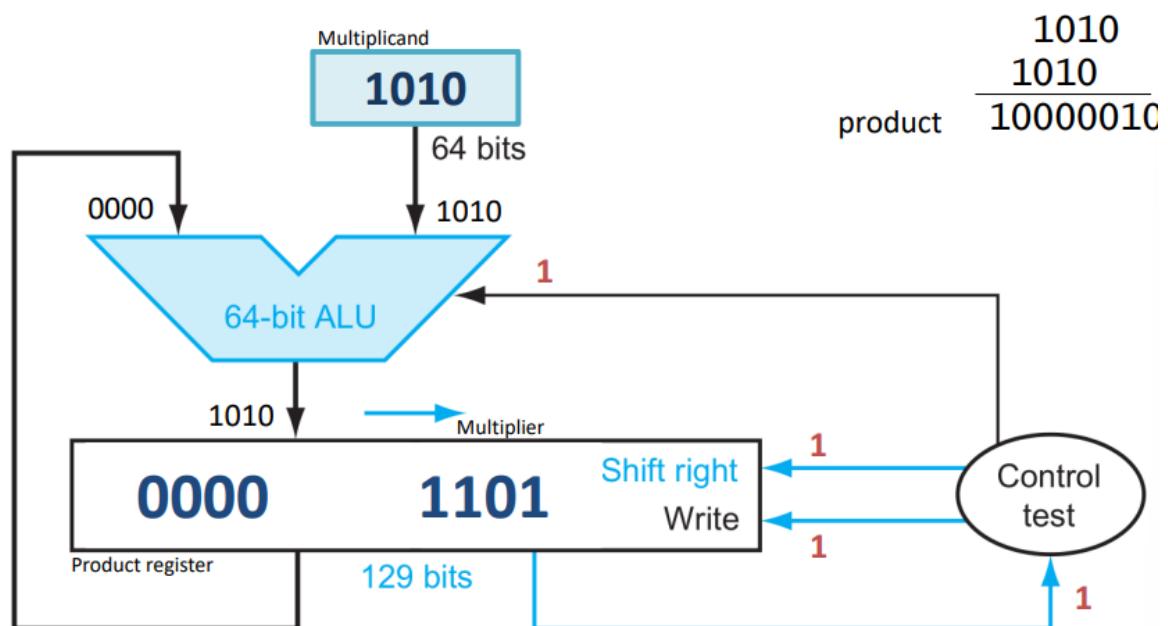
1. 초기화



- 보다시피 왼쪽에 write가 있고, 오른쪽에 Multiplier가 있다. 그리고, Product로 Control test가 Shift right, Write와 선이 연결되어 있다.
- 이제 본격적으로 연산을 시작해보자, 일단 Test LSB 를 한다. test LSB 가 Control test 에 1을 보내준다.

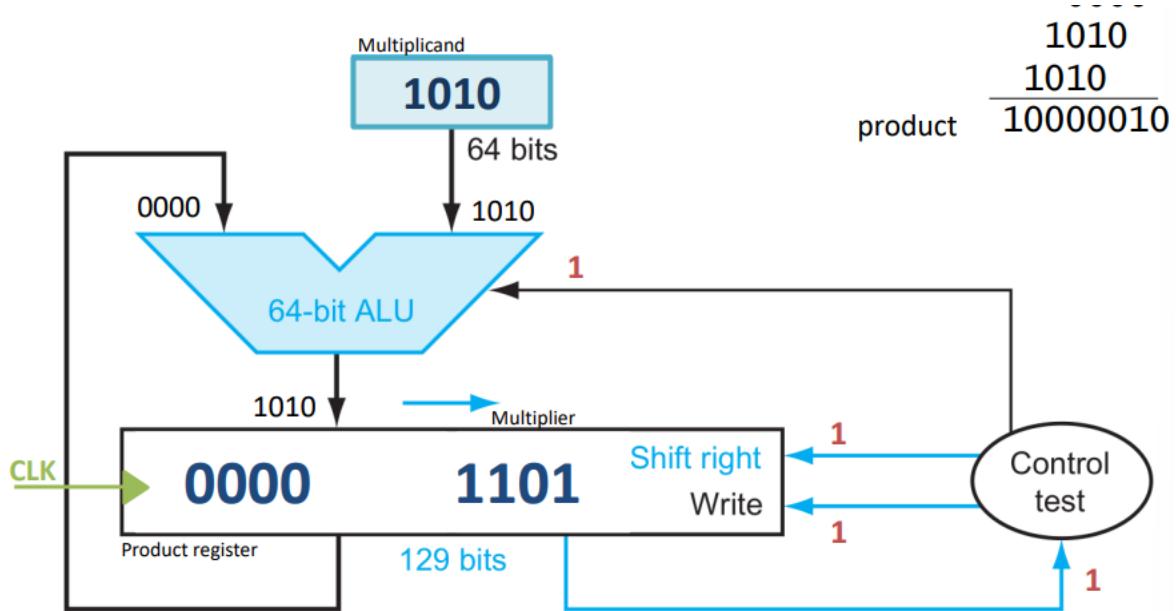


4. Control signal setup

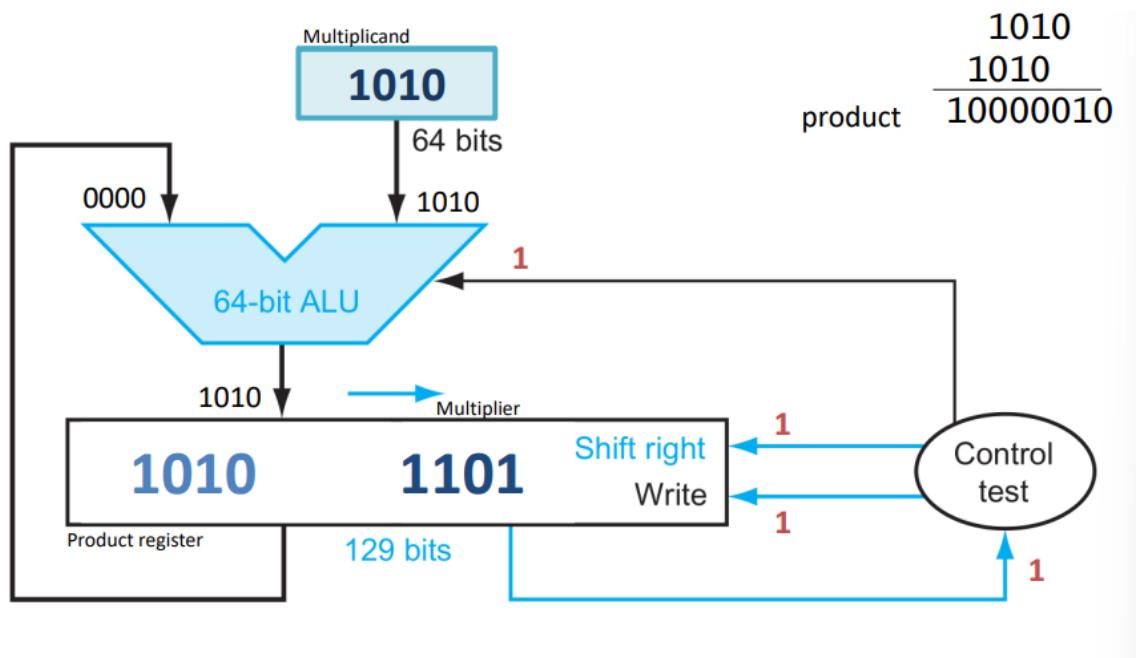


test LSB 를 한 다음, 이제 본격적으로 Control을 통해, 신호를 주려한다. 그래서 signal setup을 한다. Shift right && Write && ALU(Always 1) && Control test == 1

5. Clock firing (CLK On!) && signal maintenance

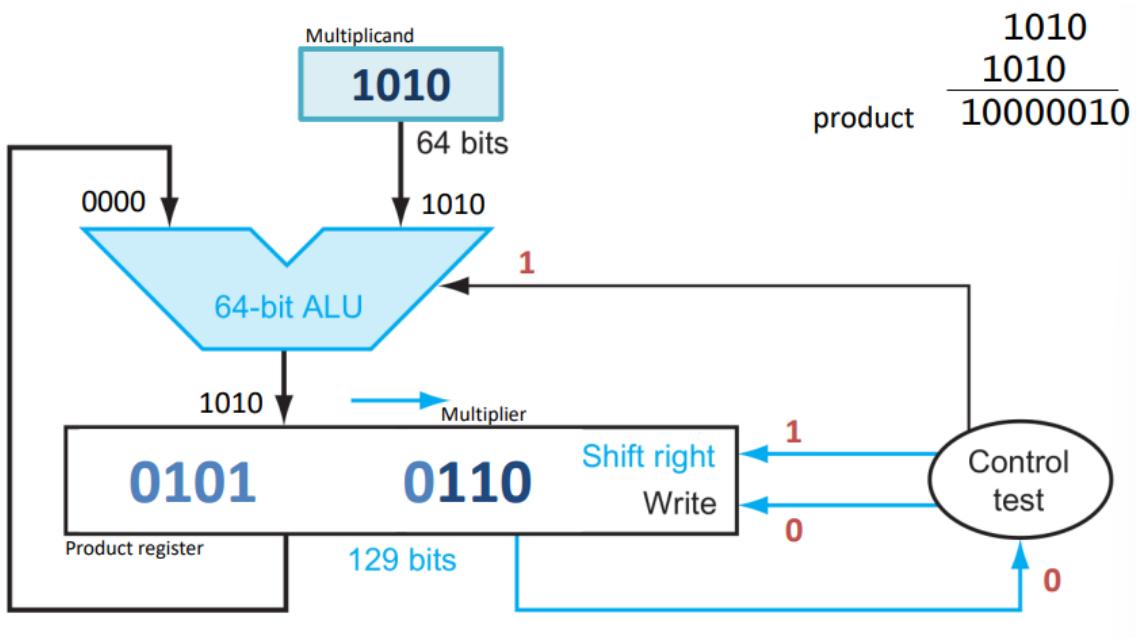


6. Multiplicand added



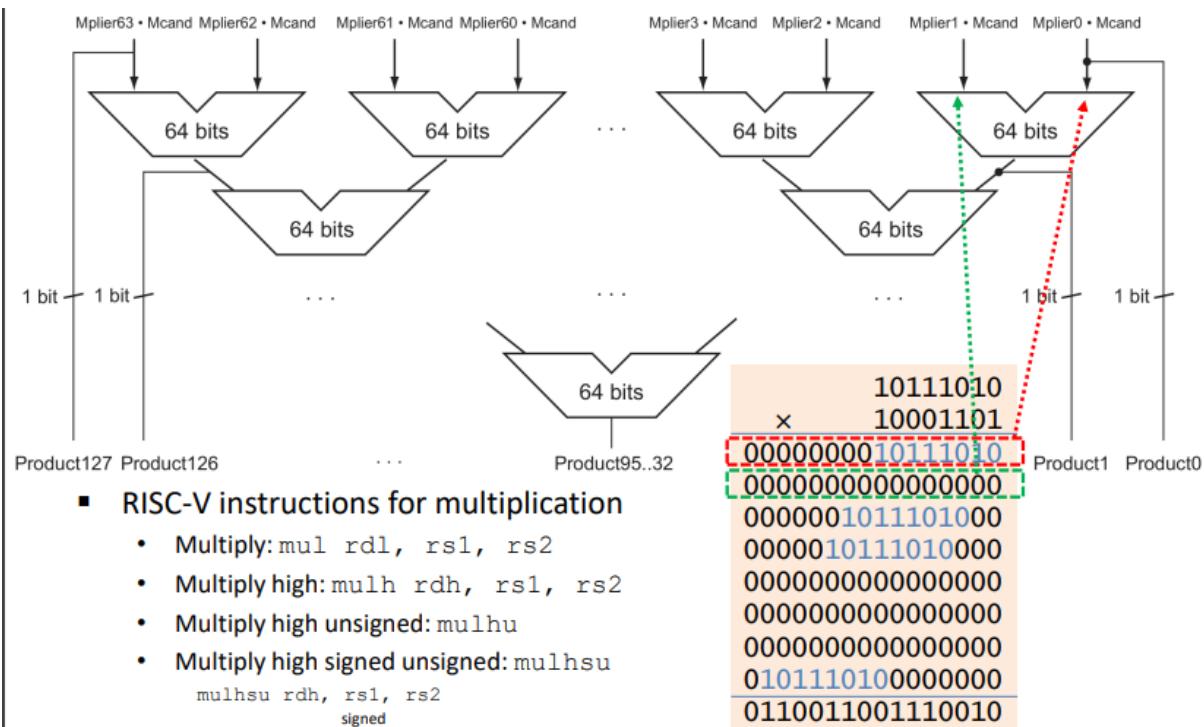
드디어 신호와 CLK가 동시에 들어와서 write에 ALU에서 연산하고 대기하고 있던 값 1010이 왼쪽에 들어오게 되었다.

7. Product register shifted



이제 write 하기 위해 1이란 신호를 줬으니, write를 했으므로, write == 0으로 바꿔준다. 그리고 test LSB == 0으로 바꿔준다. 이게 한 사이클이다. 이제 이렇게 함으로써 Multiplier 를 오른쪽으로 한 비트 밀어주면서 계속 연산을 반복하면 된다. 여기서는 3번을 더 반복해 주면 총 사이클 4번을 수행하여 Product값이 나올 수 있게 된다.

Faster Multiplication



- 여러 개의 ALU를 이용해서 한 번에 병렬적으로 많은 addition을 수행한다.
- ALU로 이루어진 트리 같은 구조
- 쌓으로 묶어 가면서 계속 addition
- 위의 예제는 3Cycle을 소요했다.

Multiplication을 하는 RISC-V Instruction

- multiply:** `mul rd1, rs1, rs2`
 - `rd1`이 64 bit까지 밖에 저장 못함 (64 bit × 64 bit의 최대 길이는 128 bit)
- multiply high:** `mulh rdh, rs1, rs2`
 - 64 bit 이상의 결과 저장 가능
- multiply high unsigned:** `mulhu`
- multiply high signed unsigned:** `mulhsu rdh, rs1, rs2`
 - `rs1`은 signed, `rs2`는 unsigned

Division

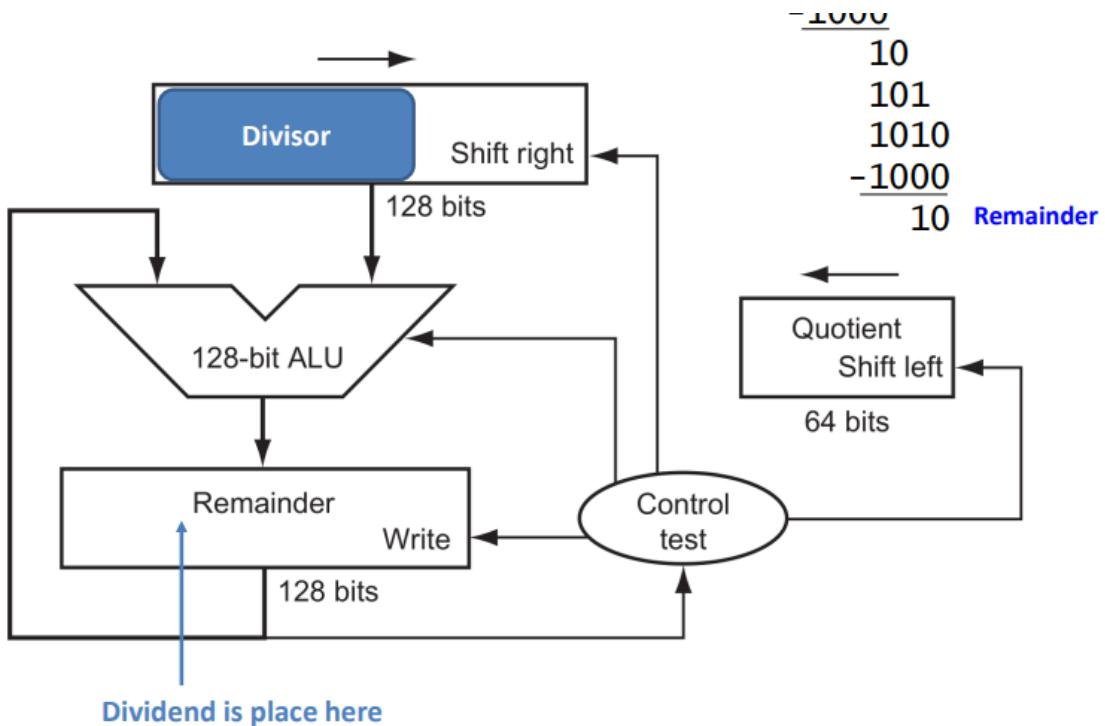
| | | |
|----------------|---------|------------------|
| Divisor | 1001 | Quotient |
| 1000) | 1001010 | Dividend |
| | -1000 | |
| | 10 | |
| | 101 | |
| | 1010 | |
| | -1000 | |
| | 10 | Remainder |

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

- Algorithm

- multiply & division 은 근본적으로 같은 operation 이라서 하나의 h/w로 multiply & division 이 모두 가능하다.
- 수행과정에서 Divisor를 빼는것은 Dividend로부터 얼마만큼 자릿수에 맞게 뺄 수 있을까 이다. 결과는 remainder reg로 간다.
- 만약 remainder $\geq 0 \rightarrow$ write 1 to quotient
- if remainder < 0 , restore by adding divisor to remainder. Write 0 to quotient.
- shift divisor right by 1 bit position
- Repeat this 33 times.

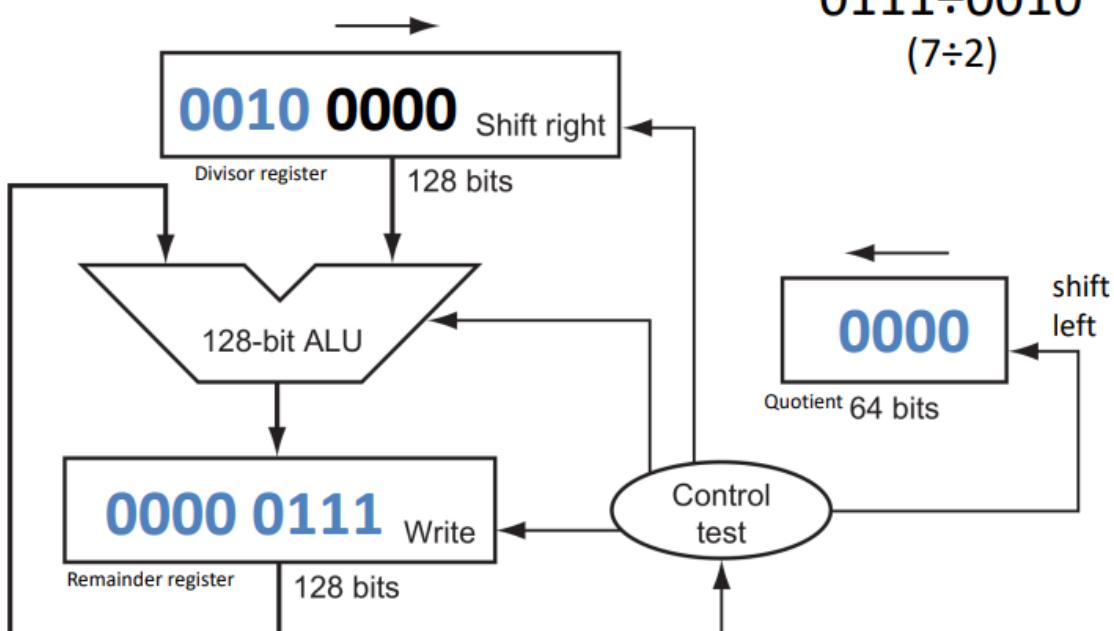
Division Hardware



- 그럼 이제 각 단계마다 한 사이클이 어떤 과정으로 이뤄지는지 확인해보자.

1. Initialization

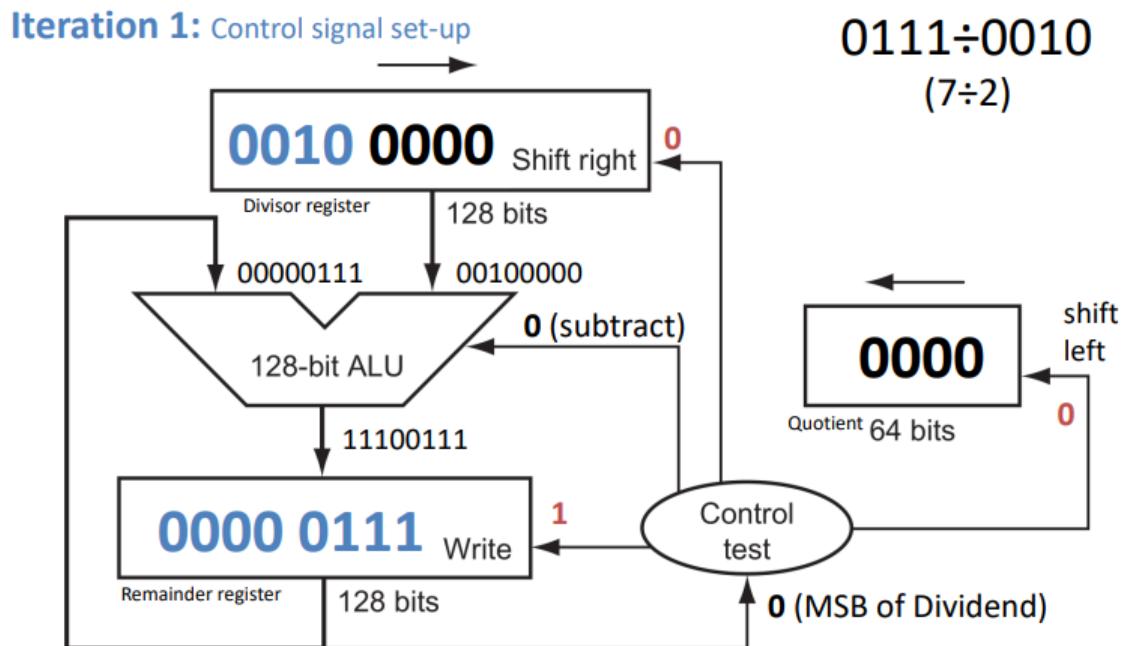
Initialization



우선 나누는 수 128비트, 나머지 128비트, 뒷 64비트에 알맞은 값을 채워넣는다. srl, sll는 곱하기와 반대인데, 회로의 구조는 동일하지만, 곱셈에서의 right은 나눗셈에서는 left가 되

었고, 그 반대의 경우도 마찬가지다.

2. Iteration 1: Control signal set-up



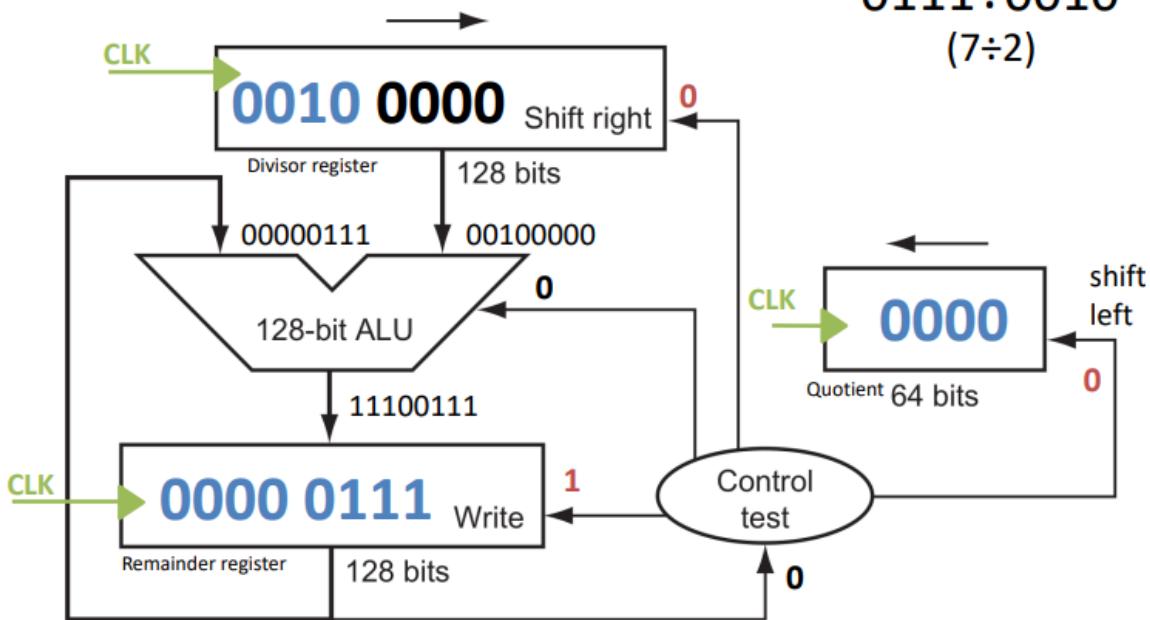
이제 반복 할 차례이니깐 신호를 주기 시작한다. 곱하기와 다르게 , 나눗셈에서는 control test에서 보내는 값이 0이면 뺄셈을 하고, 1이면 덧셈을 한다. 덧셈을 하는 경우는 나머지가 음수가 되면 덧셈을 한다.

일단 여기서는 write에 1값을 줬으므로, 아마, write에 무언가를 적기위함 인 것 같다.

3. Iteration 1: Clock firing(& Remainder updated)

Iteration 1: Clock firing (& Remainder updated)

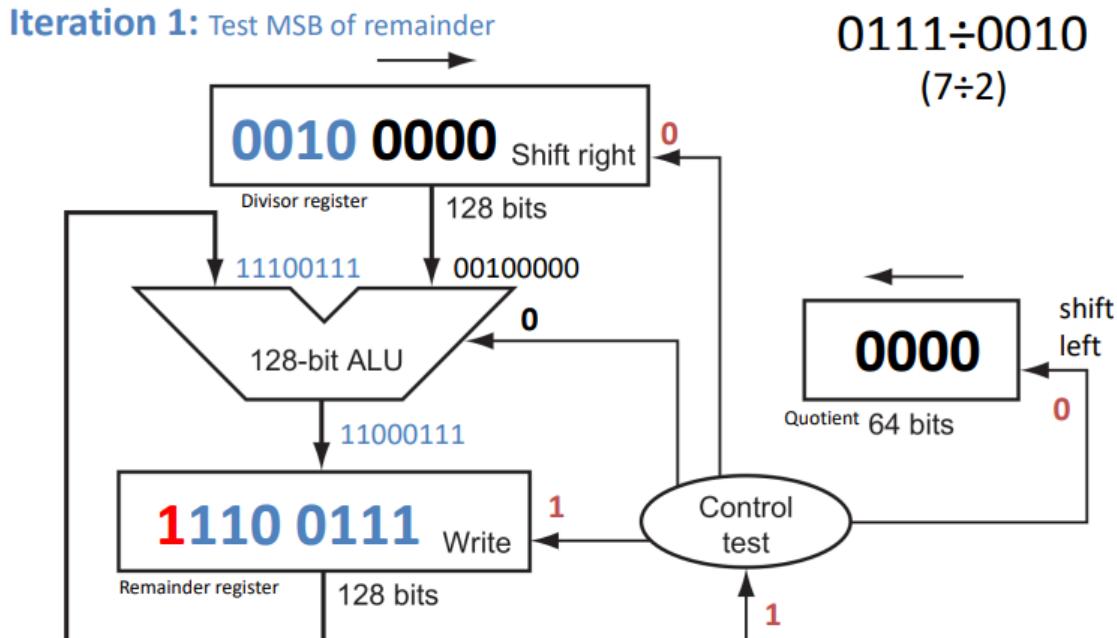
$$0111 \div 0010 \\ (7 \div 2)$$



Clock를 주었다. write의 값이 1과 CLK가 들어왔으므로 아마 다음 실행단계에서는 ALU에서 계산후에 입력 대기를 받고 있는 값인 11100111이 들어올 것이다.

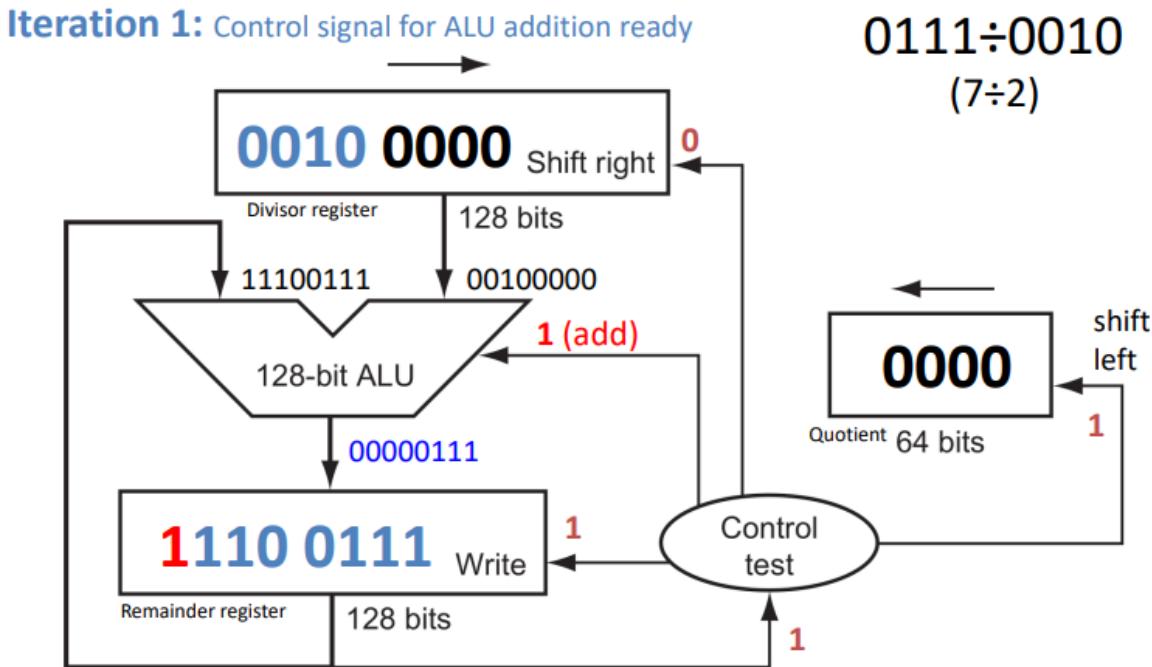
4. Iteration 1: Test MSB of remainder

$$0111 \div 0010 \\ (7 \div 2)$$



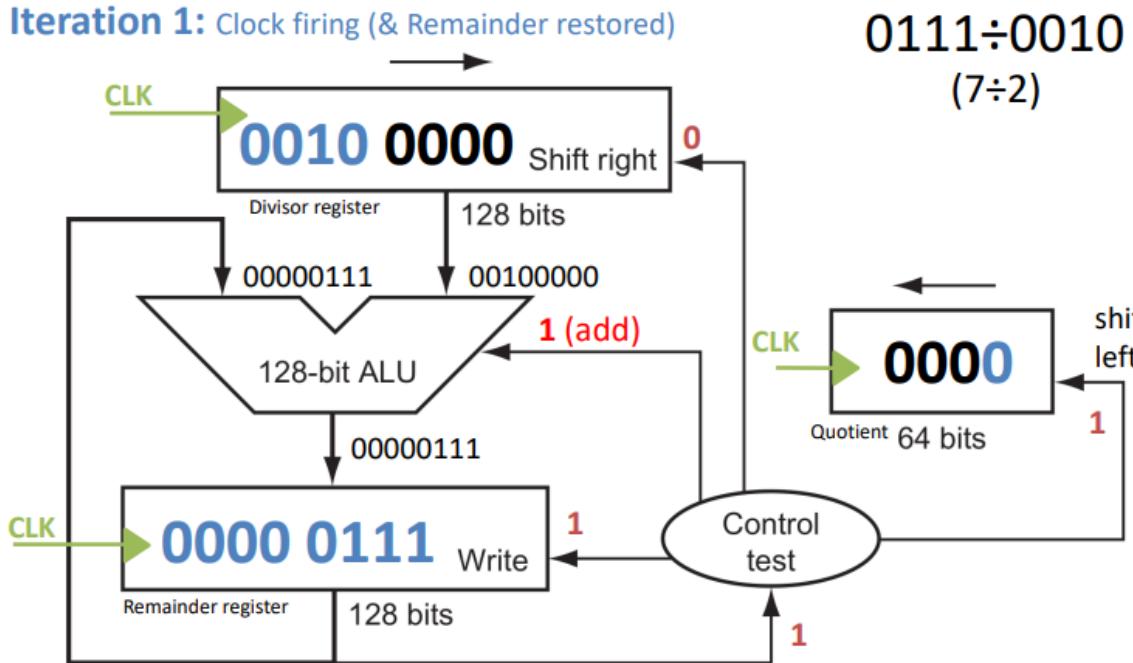
3번에서 말한 것 처럼 나머지 값이 들어왔다. 그리고 clock도 꺼졌다. 다만 문제는 나머지가 음수가 되어버렸다. 그럼 이제 ALU으로 들어가는 신호가 1(더하기) 가 될 것이다.

5. Iteration1: Control signal for ALU addition ready



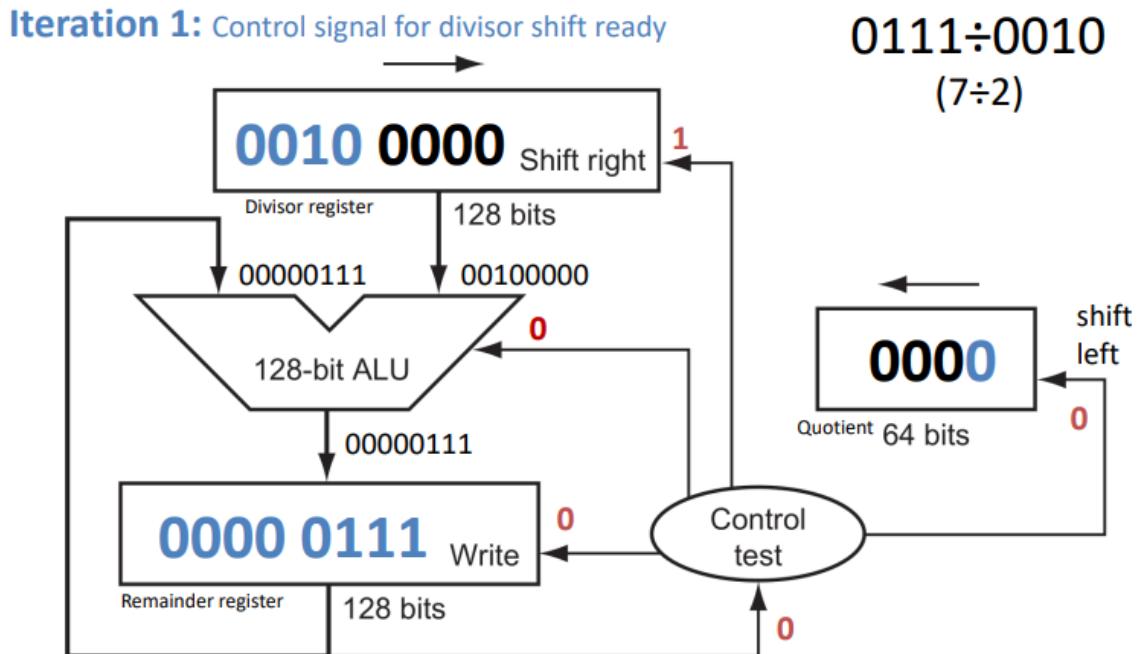
그럼 아까 전에는 뺄셈 값이 대기하고 있었지만 이제는 더하기 한 값이 대기하고 있다.

6. Iteration1 : Clock firing(& Remainder restored)



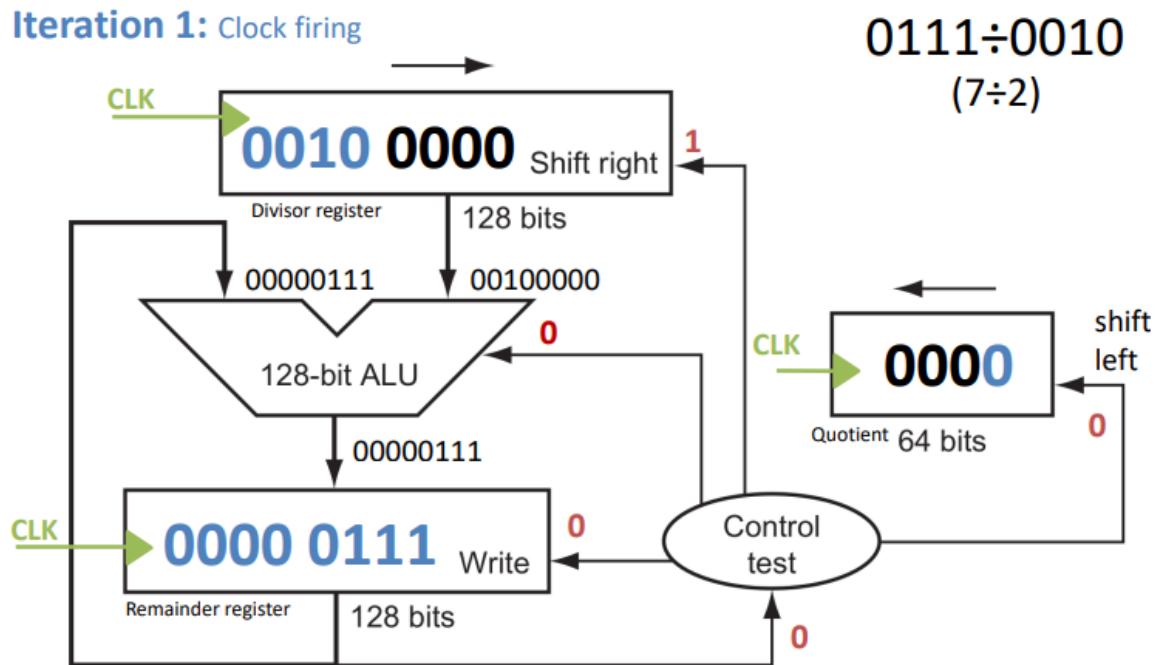
CLK가 켜졌다. 그리고 나머지 값에 신호 1이 들어왔다. 그리고, 몇의 신호에도 CLK와 1이 동시에 불이 켜졌다. 그럼 대기하고 있던 더하기 값이 나머지의 값으로 들어오게 될 것이고, 들어온 값이 또 ALU의 연산에 들어가게 될 것이다.

7. Iteration1: Control signal for divisor shift ready



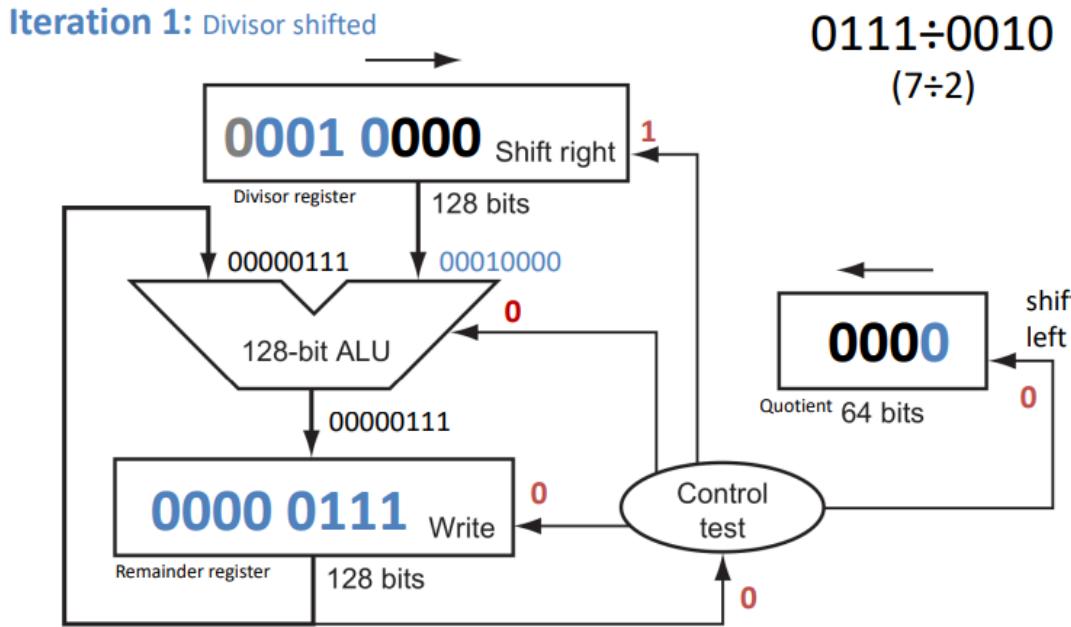
6번에서 말했던 것처럼 되었다. 이제 회로를 살펴보면, srl값에 1이란 신호가 들어왔다.

8. Iteration1 : Clock firing



srl 값에 CLK && 1 이 되었다. 그럼 srl연산이 수행이 되고, ALU연산 값에 srl값이 들어가서 나머지가 음수가 아니면 0, 음수면 1이 실행이 될 것이다.

9. Iteration 1: Divisor shifted



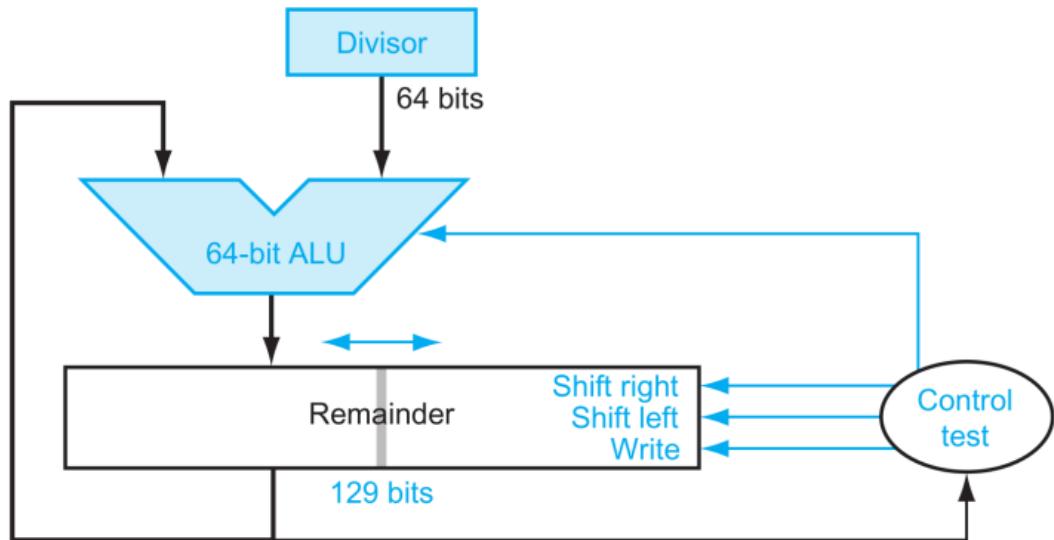
8번에서 말 했던 것처럼 되었다. $sll == CLK \&& 1$ 이면, sll 이 수행되며, 이렇게 한 사이클이 반복 될 것이다.

Division Example 7/2 (0111/0010)

| Iteration | Step | Quotient | Divisor | Remainder |
|-----------|---|----------|-----------|-----------|
| 0 | Initial values | 0000 | 0010 0000 | 0000 0111 |
| 1 | 1: Rem = Rem - Div | 0000 | 0010 0000 | 0110 0111 |
| | 2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0 | 0000 | 0010 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0001 0000 | 0000 0111 |
| 2 | 1: Rem = Rem - Div | 0000 | 0001 0000 | 0111 0111 |
| | 2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0 | 0000 | 0001 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 1000 | 0000 0111 |
| 3 | 1: Rem = Rem - Div | 0000 | 0000 1000 | 0111 1111 |
| | 2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0 | 0000 | 0000 1000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 0100 | 0000 0111 |
| 4 | 1: Rem = Rem - Div | 0000 | 0000 0100 | 0000 0011 |
| | 2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1 | 0001 | 0000 0100 | 0000 0011 |
| | 3: Shift Div right | 0001 | 0000 0010 | 0000 0011 |
| 5 | 1: Rem = Rem - Div | 0001 | 0000 0010 | 0000 0001 |
| | 2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1 | 0011 | 0000 0010 | 0000 0001 |
| | 3: Shift Div right | 0011 | 0000 0001 | 0000 0001 |

위에서 했던 사이클을 요약해 준 것이며, 전체 사이클 순서를 축약해서 나타낸 것이다.

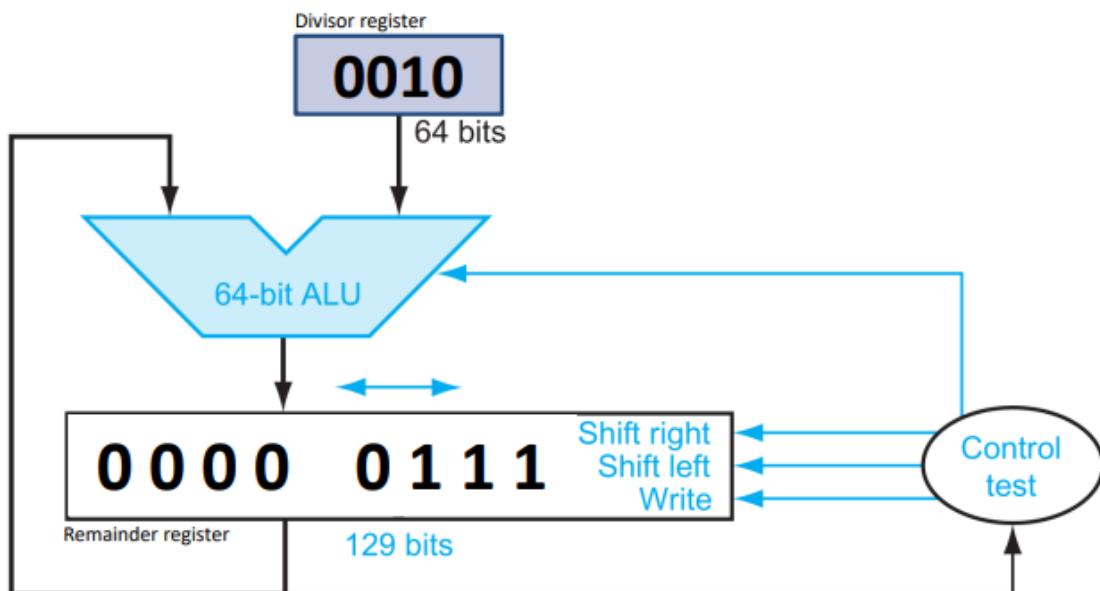
Improved Division Hardware



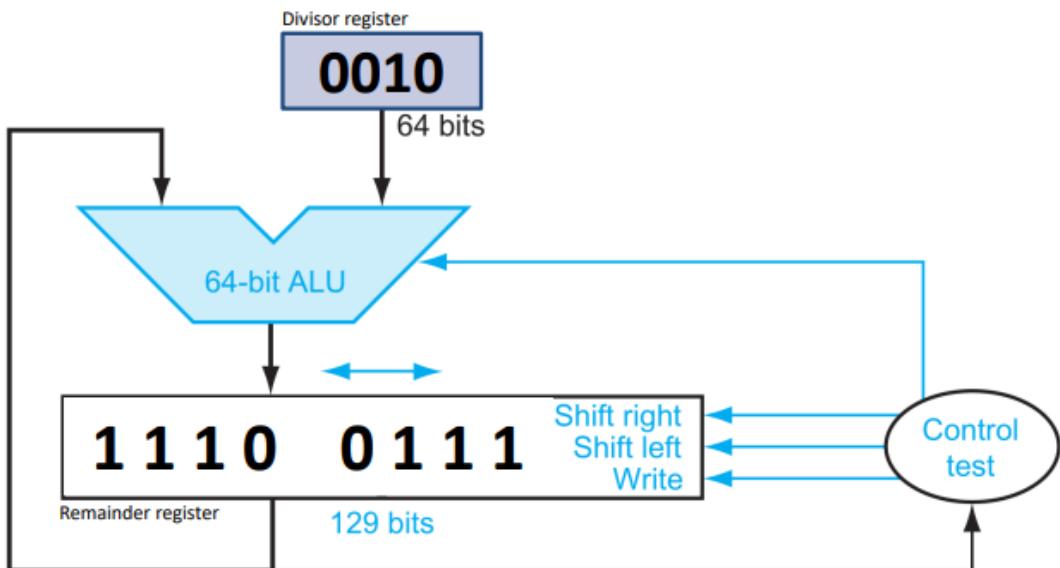
나눗셈 회로의 향상된 버전은 이렇다.

그럼, 이제부터 연산을 순차적으로 살펴보자.

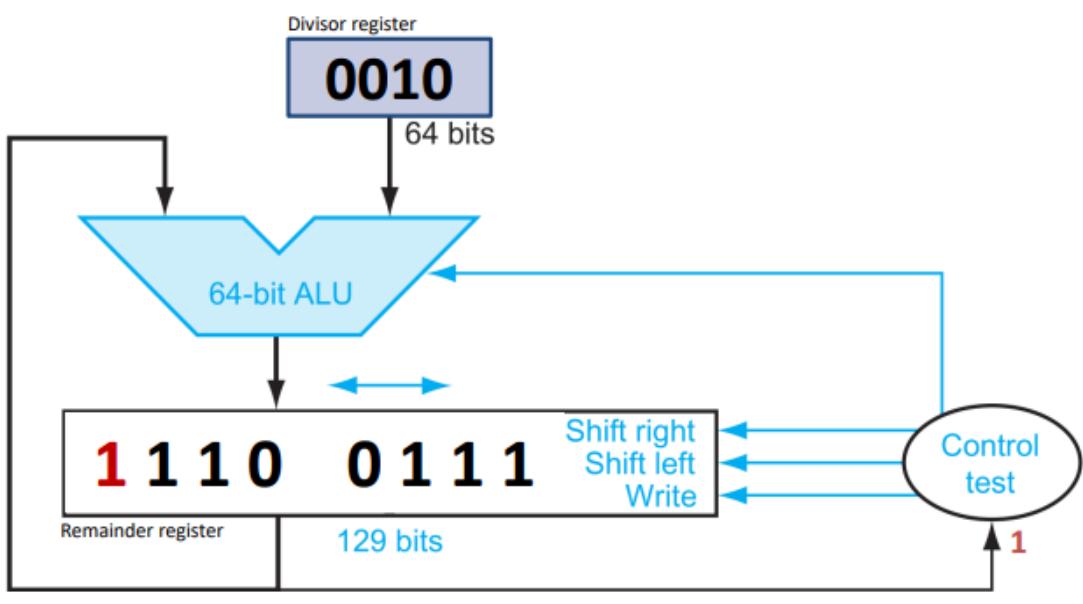
1. Iteration 0 : Setting up reg



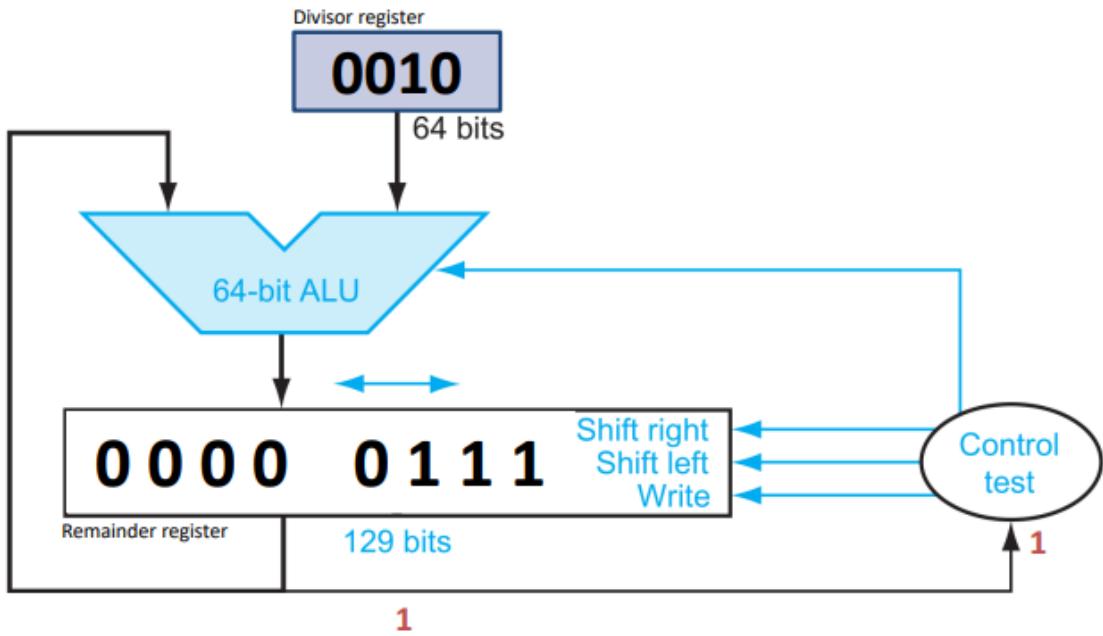
2. Iteration1 : remainder = 나머지의 왼쪽 - 나누는 수(사실 연산 같은 위에서 했던 것과 별반 다를 것이 없다.) $0000\ 0111 - 0010\ 0000 = 1110\ 0111$



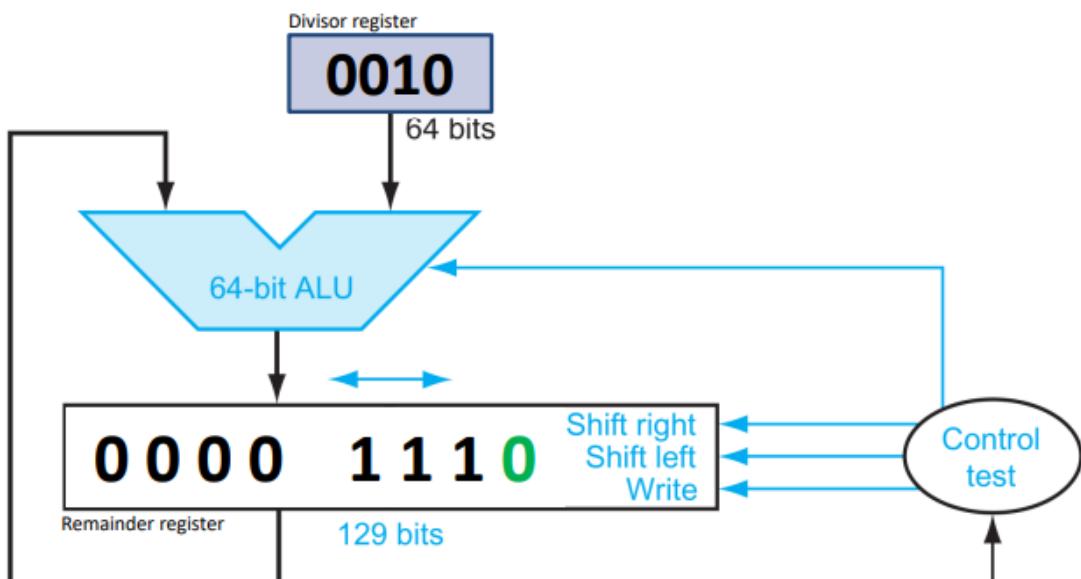
3. Iteration1 : MSB를 체크해본다. 나머지의 MSB가 1이면 그 다음 단계에선 더하기를 Remainder reg + Divisor register 을 수행한다.



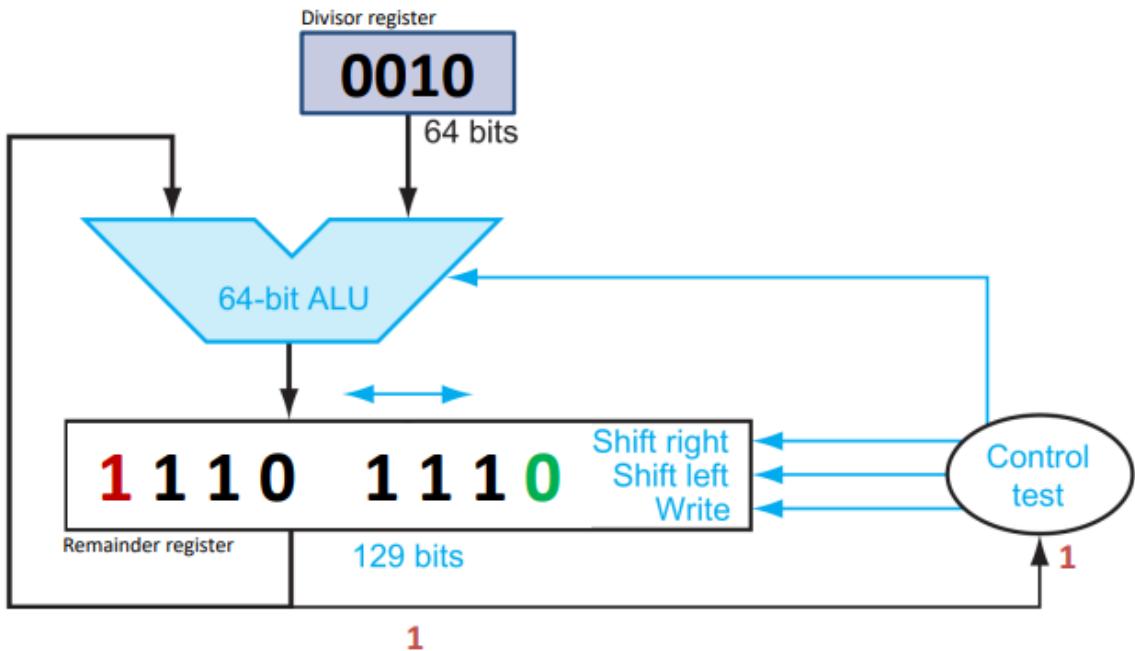
4. Iteration1 : Restore by adding the divisor back to the remainder



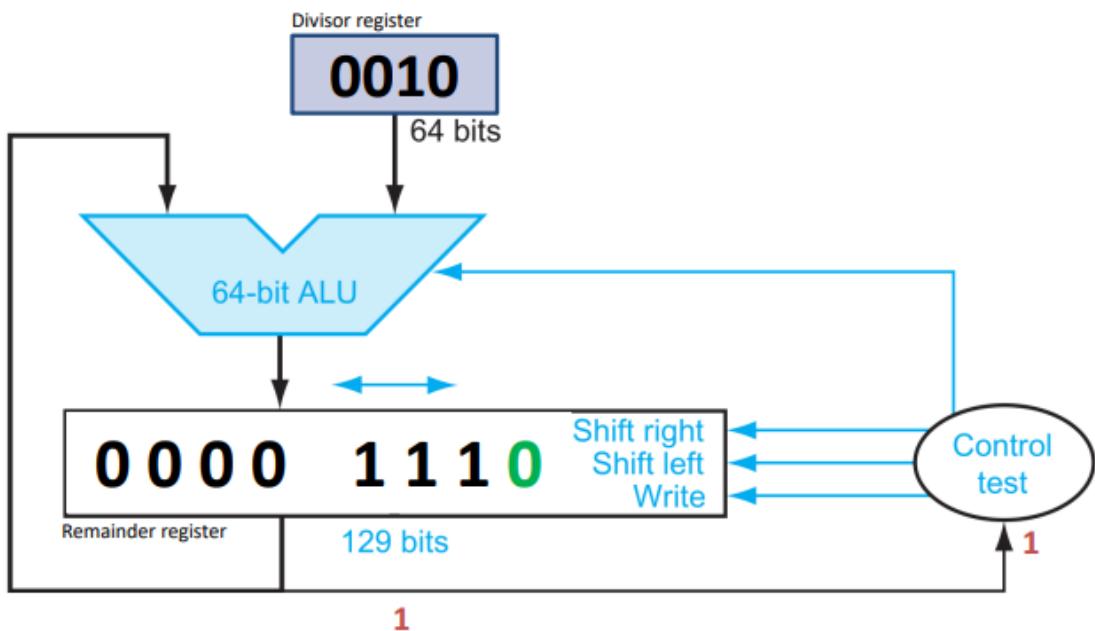
5. Iteration 1: Shift left the remainder(왼쪽으로 한 비트 옮겨준다.)



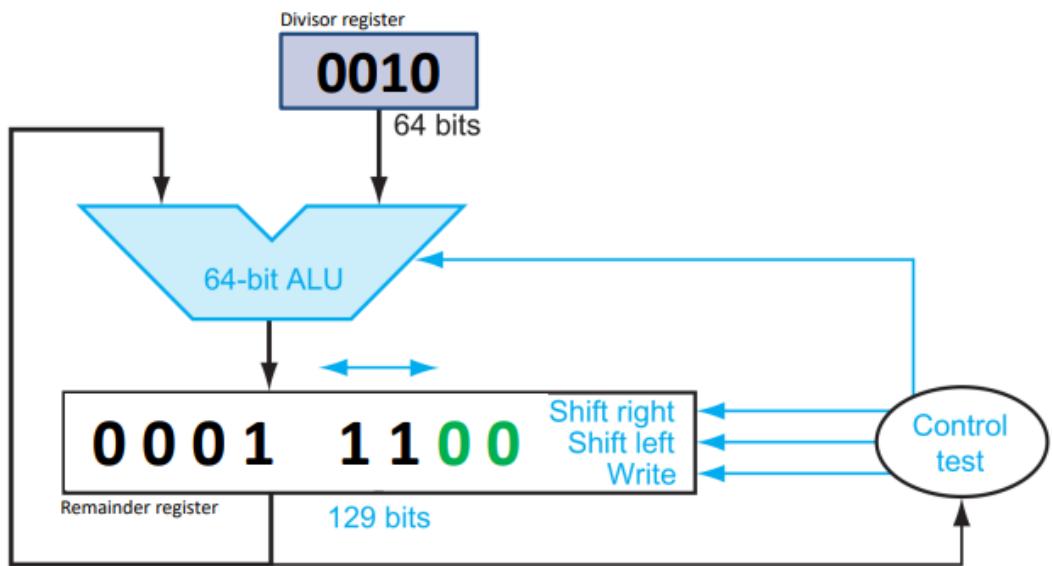
6. 비트를 옮겼으면, 해당 나머지 값 - Divisor을 해 준다.



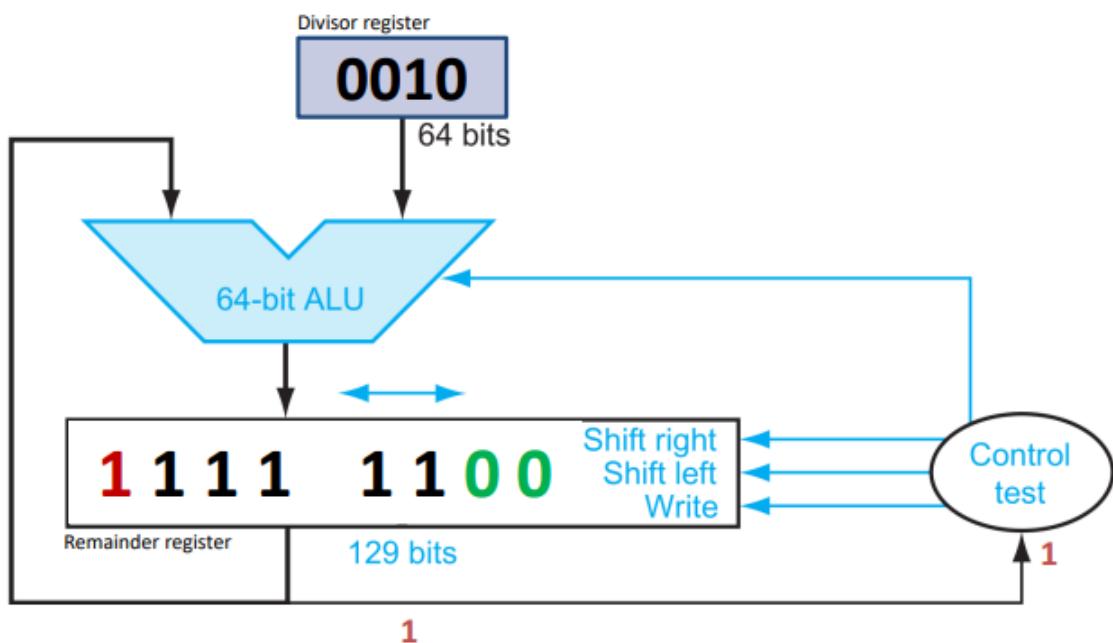
7. MSB를 체크 한 후, 1이면 더하고, 0이면 그대로 진행한다. 여기선 1 이므로 더하기를 진행해준다.



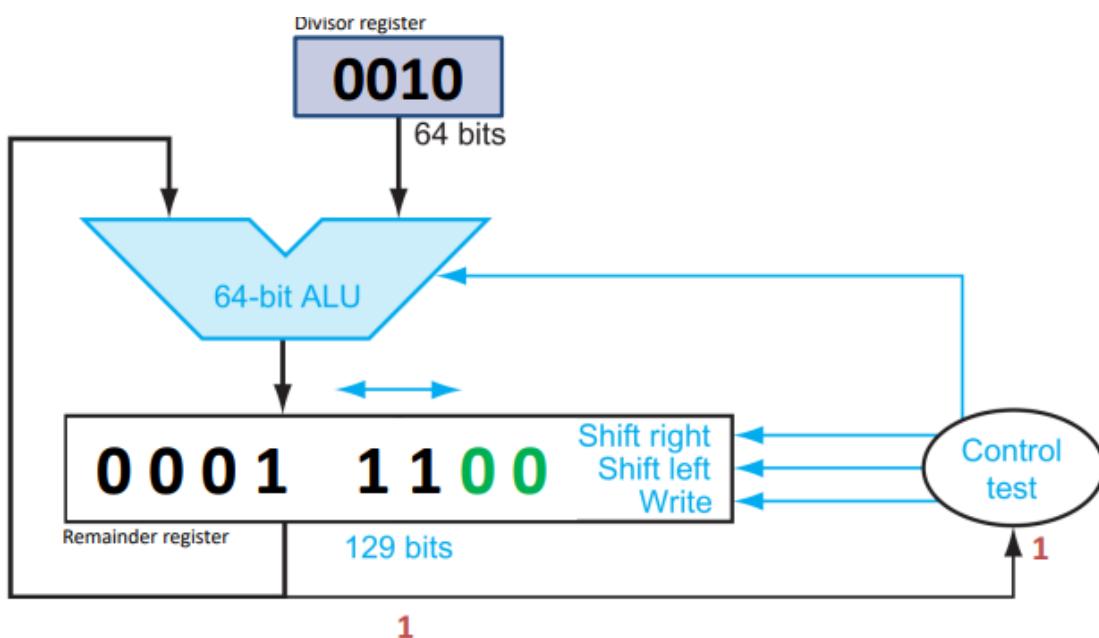
8. 그 후 remainder bit를 또 왼쪽으로 한 비트 옮겨준다.



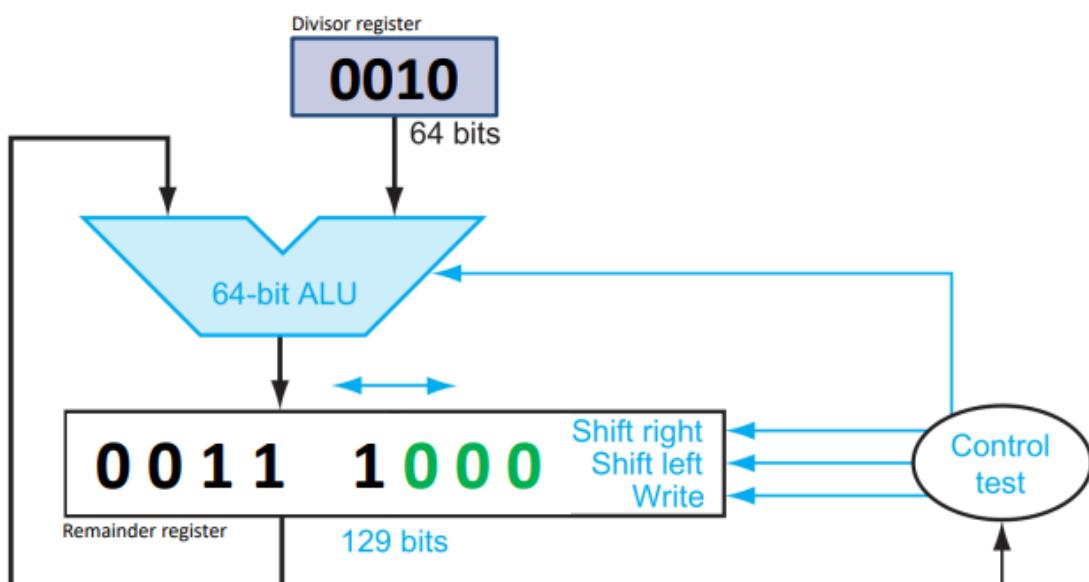
9. 그 다음 MSB check 후 0이면 빼기, 1이면 더하기 해 준다. 여기서는 또 remainder - Divisor을 해 준다.



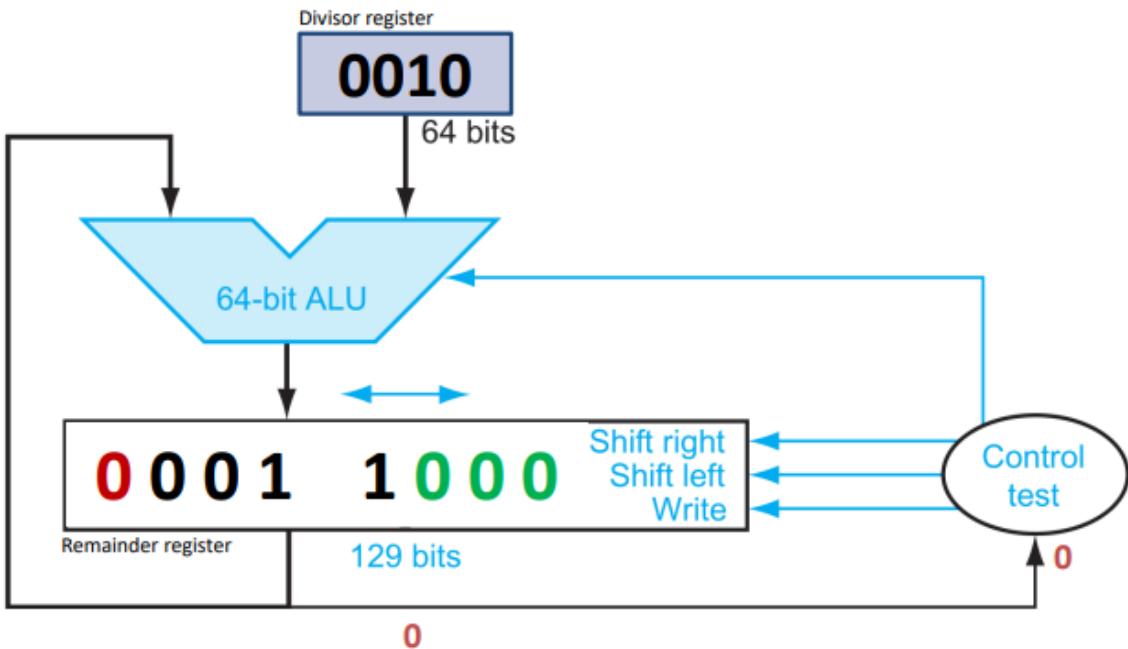
10. 연산 수행



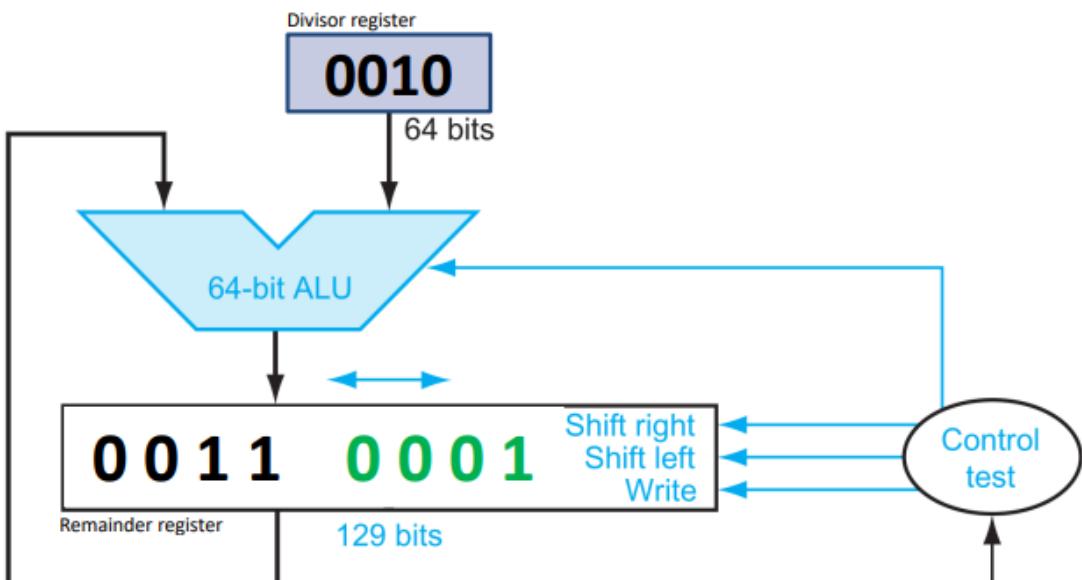
11. 왼쪽으로 한 비트 이동하기



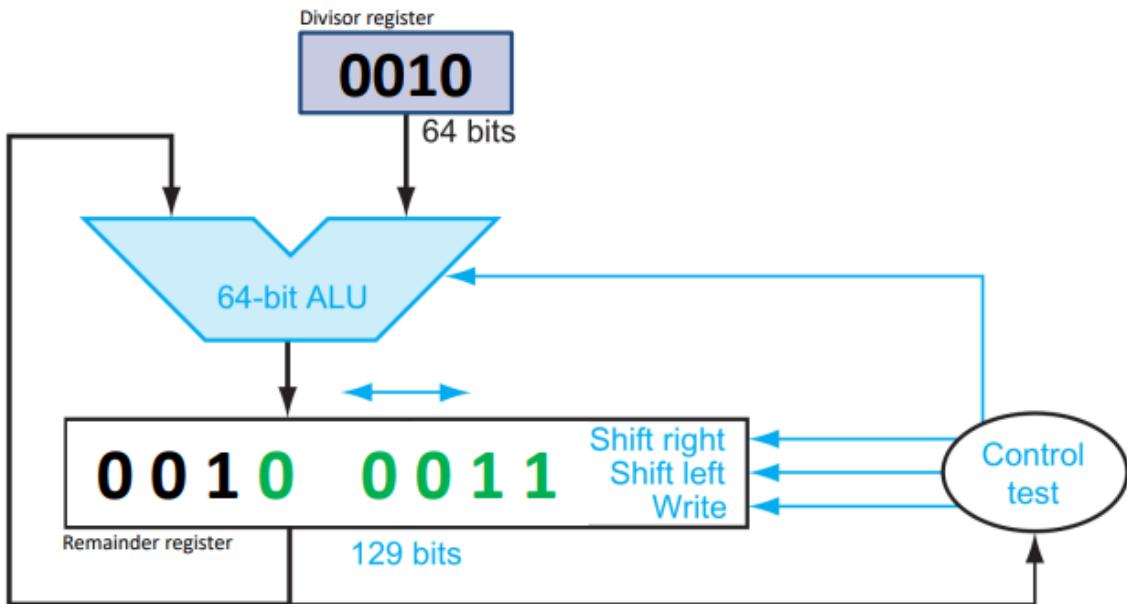
12. MSB is 0 so, we try sub



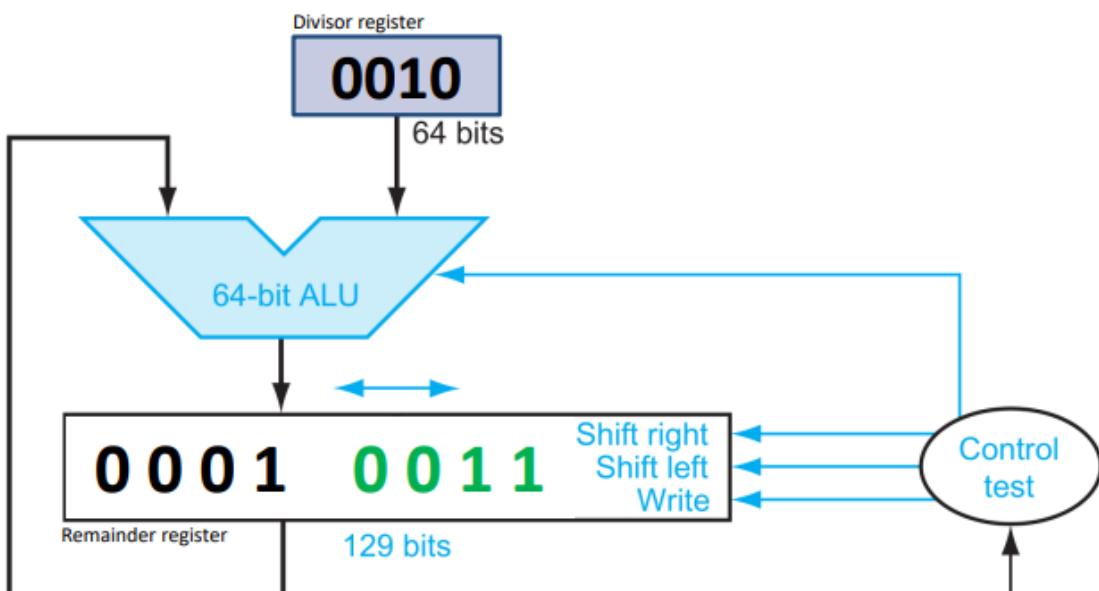
13. After number 12 product check the MSB, This case is 0, so, we try the sll, why?
이번에 sll을 하면, remainder reg의 오른편이 다 shift가 일어나기 때문이다.



14. Check the MSB and MSB == 0 then, sll



15. 마지막 단계다. 이제 왼쪽으로 계속 비트를 이동하다가 보니깐, remainder를 반으로 나눈 오른쪽 페이지까지 비트수가 넘어가게 되었다. 이제 마지막 조정단계에서는 오른쪽으로 한 비트 이동시켜주면 끝이 난다.



- Summary

| Iteration | Step | Divisor | Remainder (Quotient will place in right half) |
|-----------|--|---------|--|
| 0 | Initial Values | 0010 | 0000 : 0111 |
| 1 | 1: Rem = Rem - Div | 0010 | 1110 : 0111 |
| | 2: Rem < 0 → + Div, sll Q, Q0=0 | 0010 | 0000 : 1110 |
| 2 | 1: Rem = Rem - Div | 0010 | 1110 : 1110 |
| | 2: Rem < 0 → + Div, sll Q, Q0=0 | 0010 | 0001 : 1100 |
| 3 | 1: Rem = Rem - Div | 0010 | 1111 : 1100 |
| | 2: Rem < 0 → + Div, sll Q, Q0=0 | 0010 | 0011 : 1000 |
| 4 | 1: Rem = Rem - Div | 0010 | 0001 : 1000 |
| | 2: Rem >= 0 → sll Q, Q0=1 | 0010 | 0011 : 0001 |
| 5 | 1: Rem = Rem - Div | 0010 | 0001 : 0001 |
| | 2: Rem >= 0 → sll Q, Q0=1 | 0010 | 0010 : 0011 |
| 6 | Shift right the left half of remainder | 0010 | 0001 : 0011 |

Benefit of the Improved the Mul

가장 장점은 빼기를 저장하지 않아도 된다는 것이고, 몫을 Remainder의 오른쪽에 적을 수 있다. 그래서 몫이랑 나머지를 따로 저장할 공간을 2개 만들지 않고, 129비트의 용량에 sll 을 하면서 같이 저장 할 수 있게 되었다.

