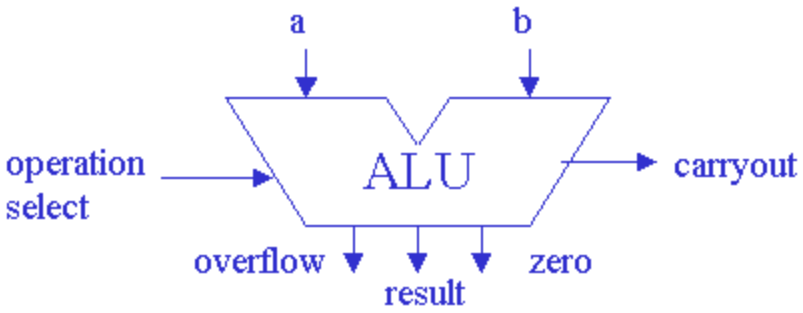
Summary of previous lecture

- number representation: usually two's complement, but other representations possible
- choice depends on hardware size, performance, development time, availability of design tools etc
- ALU:



- develop efficient, regular bit-level design from obvious word-level design
- size/performance trade-offs in different architectures e.g. ripple-carry adder versus carry-select adder

Multiply and divide

(3rd Ed: p.250-274, 4th Ed: p.230-242, 5th Ed: p.183-196)

- implementing multiplication using ALU
- Booth's multiplication algorithm
- implementing division using ALU
- related MIPS instructions
  - mult, multu, div, divu, mfhi, mflo, sll, srl, sra

Multiply: example

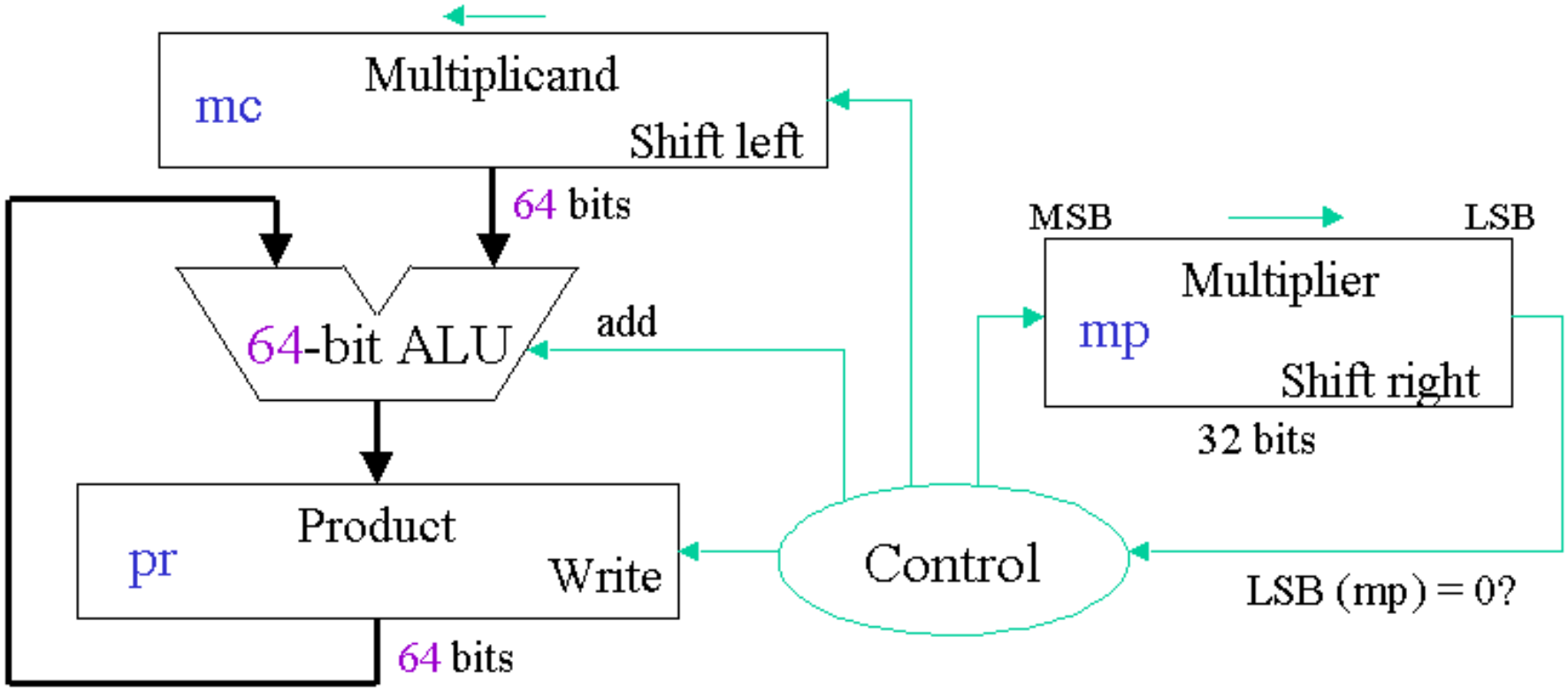
- multiplicand × multiplier = product

  2 × 11 = 22

- idea: sum of multiplicand shifted successively by 1 bit relative to multiplier; CSAA

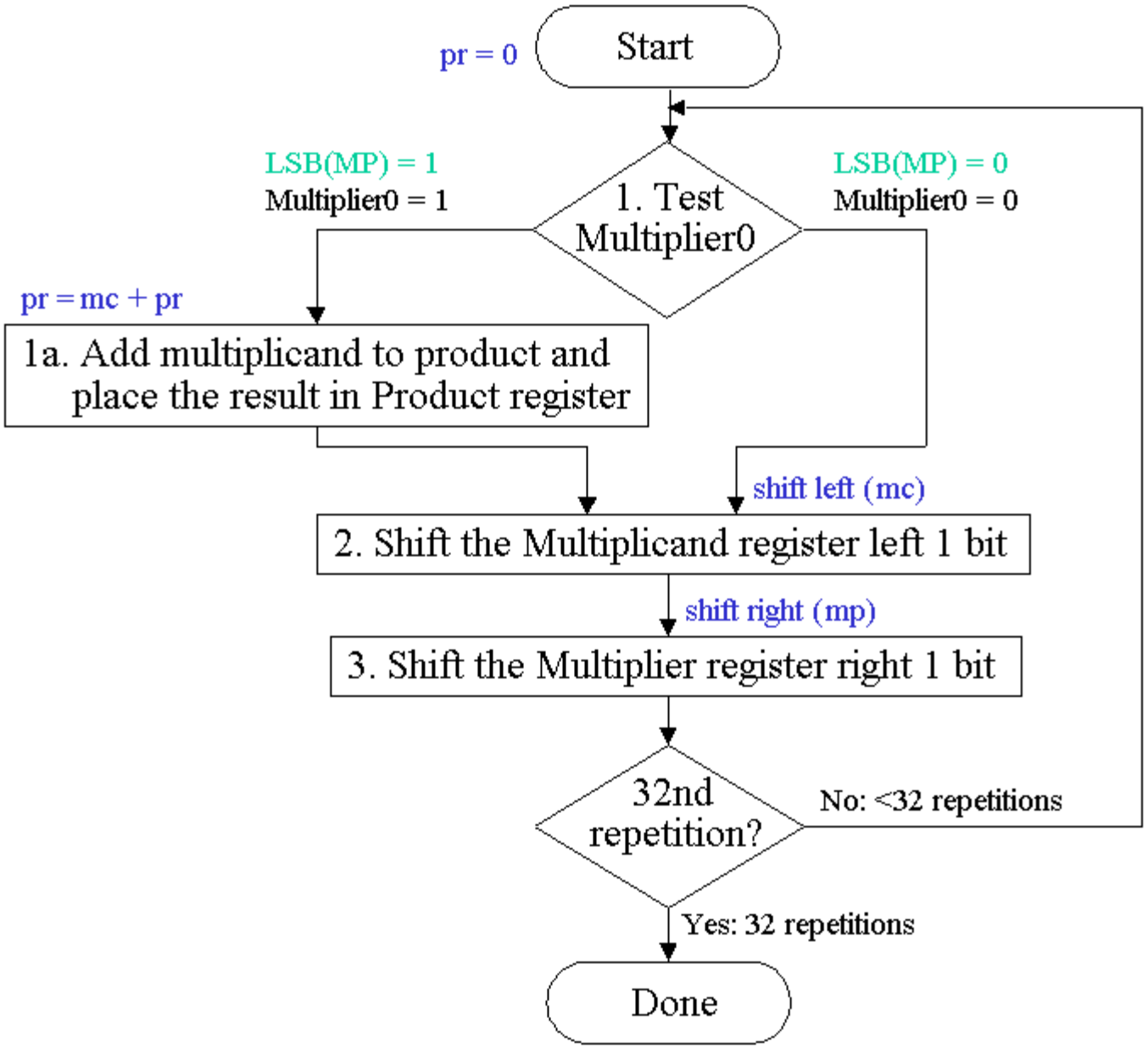
- ```
        0010 mc
  ×     1011 mp
  ---------------------------------------------
     ...0010 ← mc shifted 0 bit × bit 0 of mp
     ..0010. ←  "    "    1 " ×  " 1    "
  + 0010... ←  "    "    3 " ×  " 3    "
  ---------------------------------------------
    0010110
  ```
  ```

Multiplication hardware: first version



The multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. The 32-bit multiplicand starts in the right half of the Multiplicand register, and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialised to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

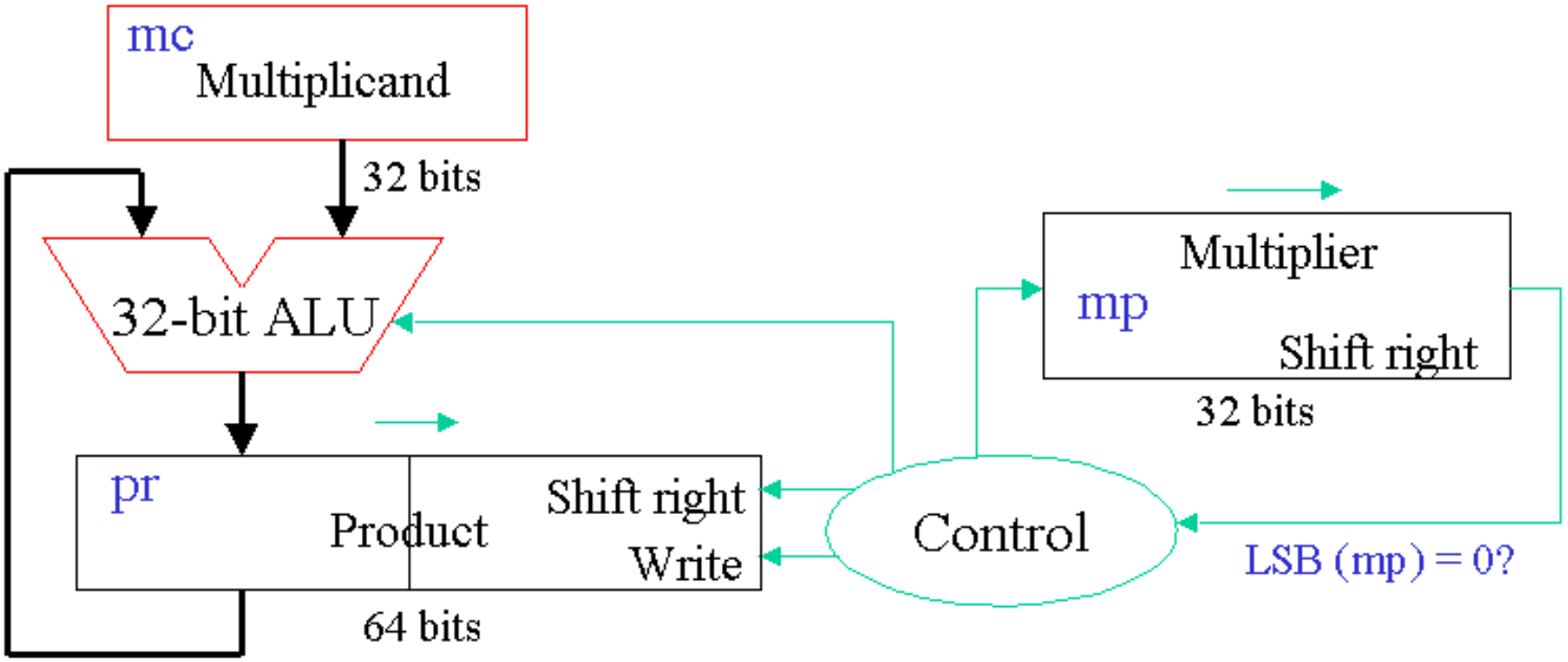
pr = mp × mc


pr = 0
Start
LSB(MP) = 1
Multiplier0 = 1
1. Test Multiplier0
LSB(MP) = 0
Multiplier0 = 0
pr = mc + pr
1a. Add multiplicand to product and place the result in Product register
shift left (mc)
2. Shift the Multiplicand register left 1 bit
shift right (mp)
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <32 repetitions
Yes: 32 repetitions
Done

Multiply example using first algorithm

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	001**1**	0000 0010	0000 0000
1	1a: **1** => Prod=Prod+Mcand 2: Shift left Multiplicand 3: Shift right Multiplier	0011 0011 000**1**	0000 0010 0000 0100 0000 0100	0000 0010 0000 0010 0000 0010
2	1a: **1** => Prod=Prod+Mcand 2: Shift left Multiplicand 3: Shift right Multiplier	0001 0001 000**0**	0000 0100 0000 1000 0000 1000	0000 0110 0000 0110 0000 0110
3	1: **0** => no operation 2: Shift left Multiplicand 3: Shift right Multiplier	0000 0000 000**0**	0000 1000 0001 0000 0001 0000	0000 0110 0000 0110 0000 0110
4	1: **0** => no operation 2: Shift left Multiplicand 3: Shift right Multiplier	0000 0000 0000	0001 0000 0010 0000 0010 0000	0000 0110 0000 0110 0000 0110

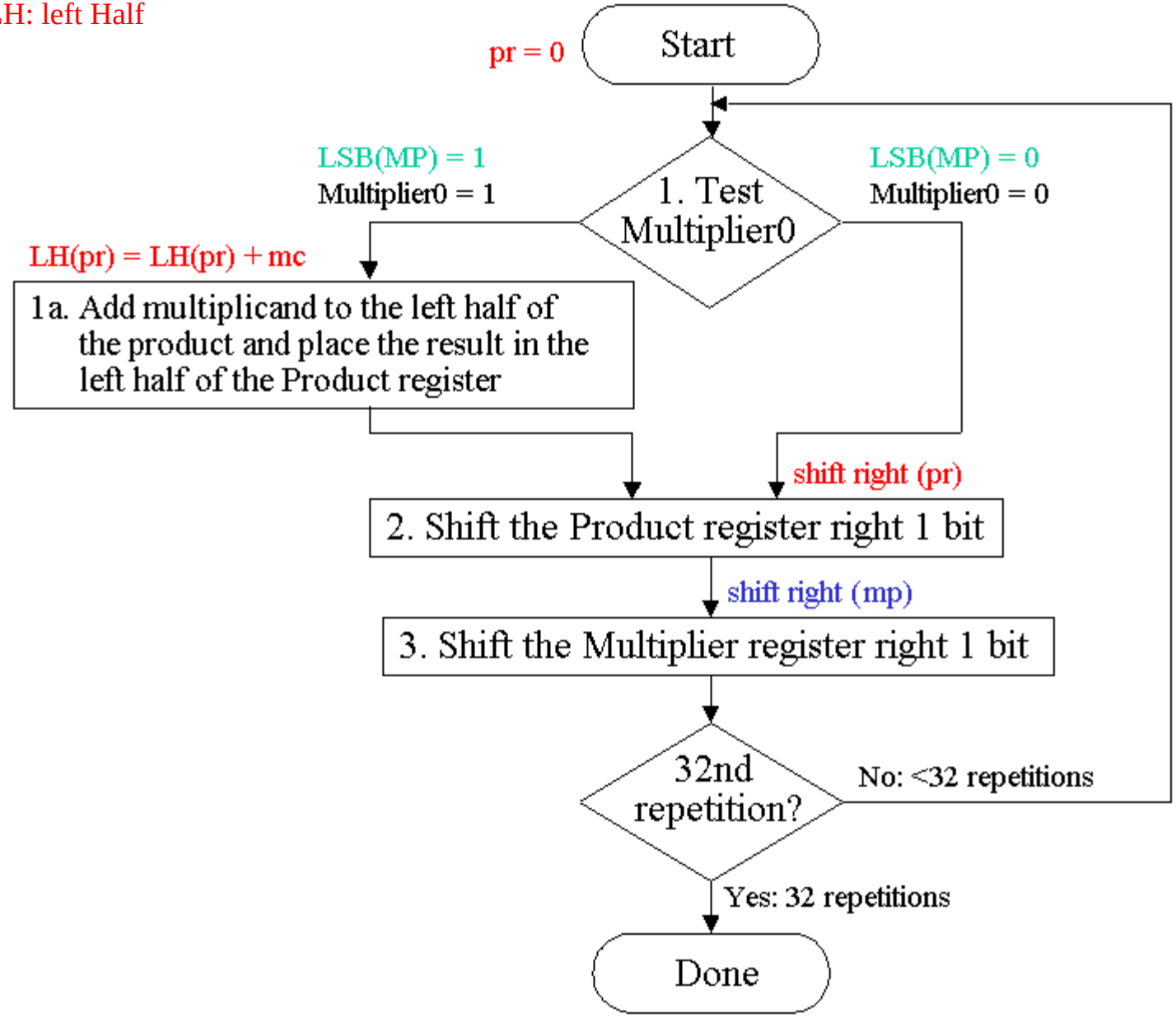
Multiplication hardware: second version



The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left as 64 bits. Now the product is shifted right.

pr = mp × mc

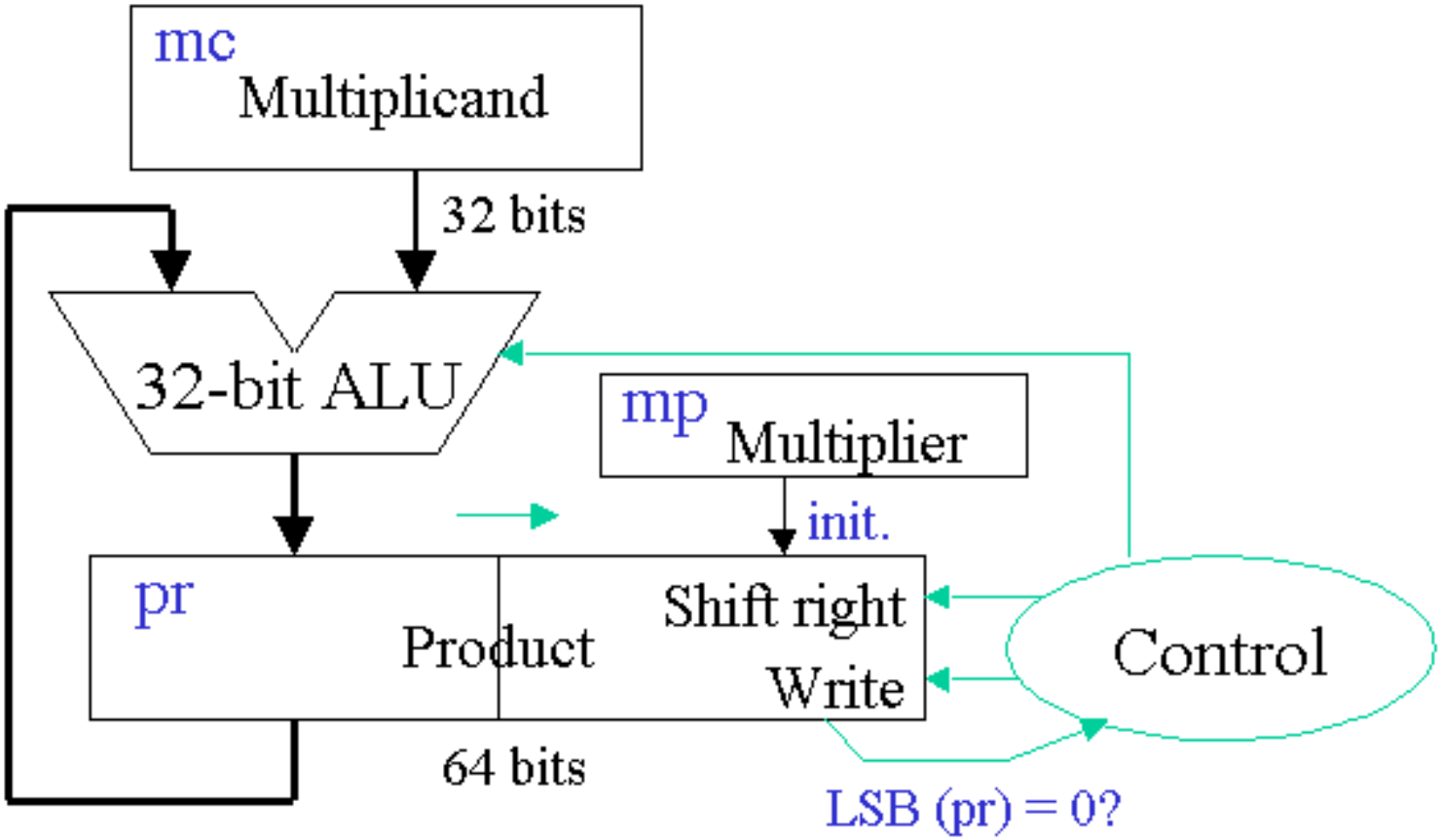
LH: left Half



Multiply example using second algorithm

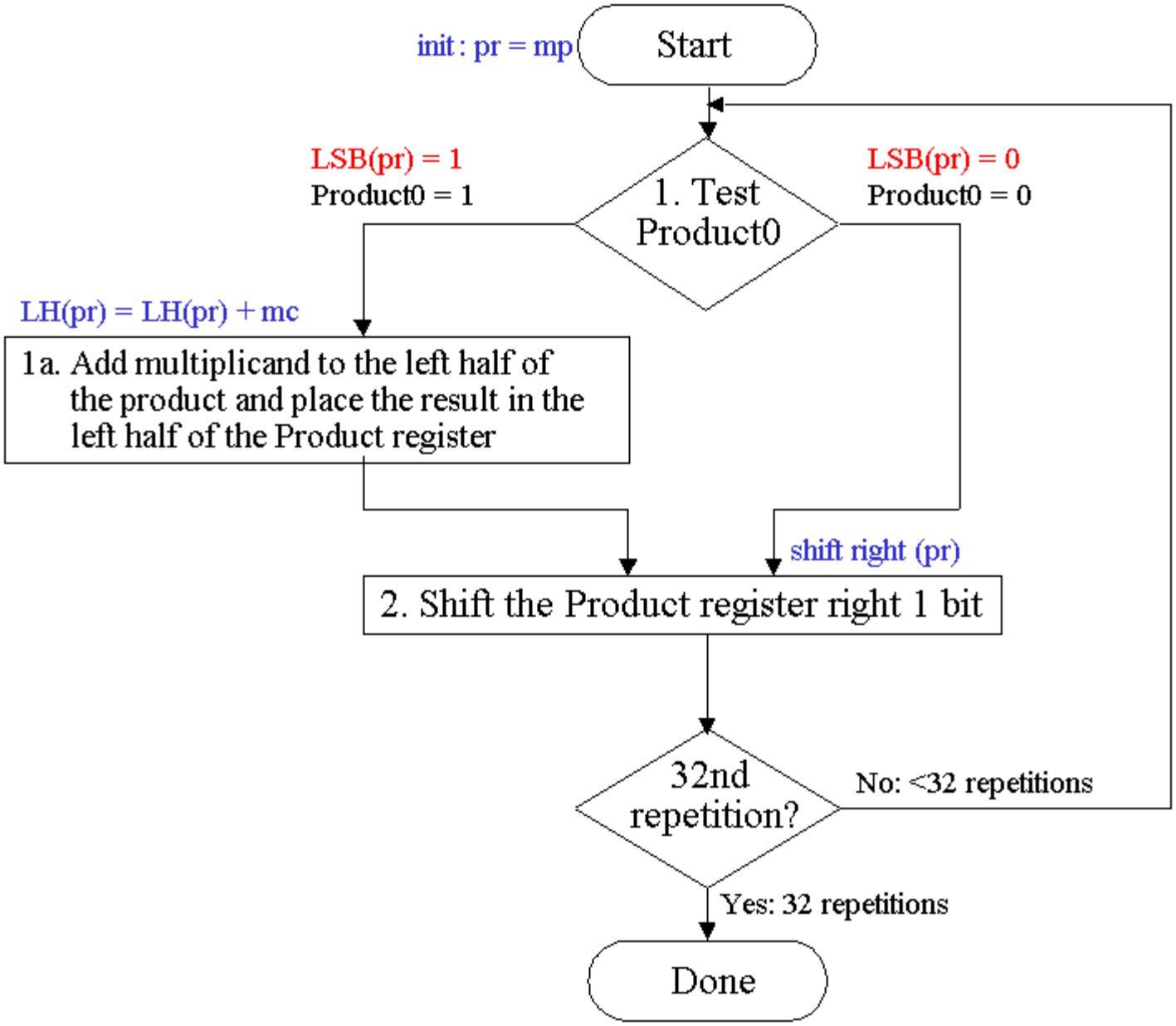
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	001**1**	0010	0000 0000
1	1a: **1** => Prod=Prod+Mcand 2: Shift right Product 3: Shift right Multiplier	0011 0011 000**1**	0010 0010 0010	0010 0000 0001 0000 0001 0000
2	1a: **1** => Prod=Prod+Mcand 2: Shift right Product 3: Shift right Multiplier	0001 0001 000**0**	0010 0010 0010	0011 0000 0001 1000 0001 1000
3	1: **0** => no operation 2: Shift right Product 3: Shift right Multiplier	0000 0000 000**0**	0010 0010 0010	0001 1000 0000 1100 0000 1100
4	1: **0** => no operation 2: Shift right Product 3: Shift right Multiplier	0000 0000 0000	0010 0010 0010	0000 1100 0000 0110 0000 0110

Multiplication hardware: third version



The separate Multiplier register disappeared. The multiplier is placed instead in the right half of the Product register.

pr = mp × mc


init : pr = mp
Start
LSB(pr) = 1
Product0 = 1
1. Test Product0
LSB(pr) = 0
Product0 = 0
LH(pr) = LH(pr) + mc
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
shift right (pr)
2. Shift the Product register right 1 bit
32nd repetition?
No: <32 repetitions
Yes: 32 repetitions
Done

Multiply example using third algorithm

Iteration	Step	Multiplicand	Product
0	Initial Values	0010	0000 001**1**
1	1a: **1** => Prod=Prod+Mcand 2: Shift right Product	0010 0010	0010 0011 0001 000**1**
2	1a: **1** => Prod=Prod+Mcand 2: Shift right Product	0010 0010	0011 0001 0001 100**0**
3	1: **0** => no operation 2: Shift right Product	0010 0010	0001 1000 0000 110**0**
4	1: **0** => no operation 2: Shift right Product	0010 0010	0000 1100 0000 0110

Booth's insight

- substitute n additions by 1 subtraction, 1 addition
- successive 1s in multiplier mp
  $\Rightarrow$ successive addition of shifted multiplicand mc

```
                0010   mc
×               0110   mp
-------------------------------------------
m = 00100 ⟶     0010   shift left mc since mp1 = 1
k = 2       +   0010   shift left mc since mp2 = 1
-------------------------------------------
            001100
```

- given number of 1s in mp = k, and initially shifted mc = m, then summing the k terms give:
  $m + 2m + 2^2m + \cdots + 2^{k-1}m$ (geometric series)

Booth's algorithm

- replace summing k terms $m + 2m + \cdots + 2^{k-1}m$ by 1 subtraction and 1 addition: $-m + 2^k m$ (and k shifts to get the 2^k factor)
- proof: let $S = 1+2+\cdots+2^{k-1}$, so $2\times S = 2+4+\cdots+2^{k-1}+2^k$
  $S = 2\times S - S = 2^k+(2^{k-1}-2^{k-1})+\cdots+(2-2)-1 = 2^k-1$
- exercise: check that it works for signed numbers
- algorithm detects a string of 1s in mp:

4 cases .. 0 1 1 1 1 .. 1 1 0 0 0

Comparing the third algorithm and Booth's algorithm for positive numbers

Iteration	Multi-plicand	Original algorithm Step	Original algorithm Product	Booth's algorithm Step	Booth's algorithm Product
0	0010	Initial Values	0000 011**0**	Initial Values	0000 011**0 0**
1	0010 0010	1: 0 => no operation 2: Shift right Product	0000 0110 0000 001**1**	1a: **00** => no operation 2: Shift right Product	0000 0110 0 0000 001**1 0**
2	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0010 0011 0001 000**1**	1c: **10** => Prod=Prod-Mcand 2: Shift right Product	1110 0011 0 1111 000**1 1**
3	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0011 0001 0001 100**0**	1d: **11** => no operation 2: Shift right Product	1111 0001 1 1111 000**0 1**
4	0010 0010	1: 0 => no operation 2: Shift right Product	0001 1000 0000 1100	1b: **01** => Prod=Prod+Mcand 2: Shift right Product	0001 1000 1 0000 1100 0

case
- 00: middle of a string of 0s, no action
- 01: end of a string of 1s, pr = pr + shifted mc
- 10: start of a string of 1s, pr = pr - shifted mc
- 11: middle of a string of 1s, no action

then shift right pr

Division

- invented by Briggs
- Dividend = Quotient × Divisor + Remainder

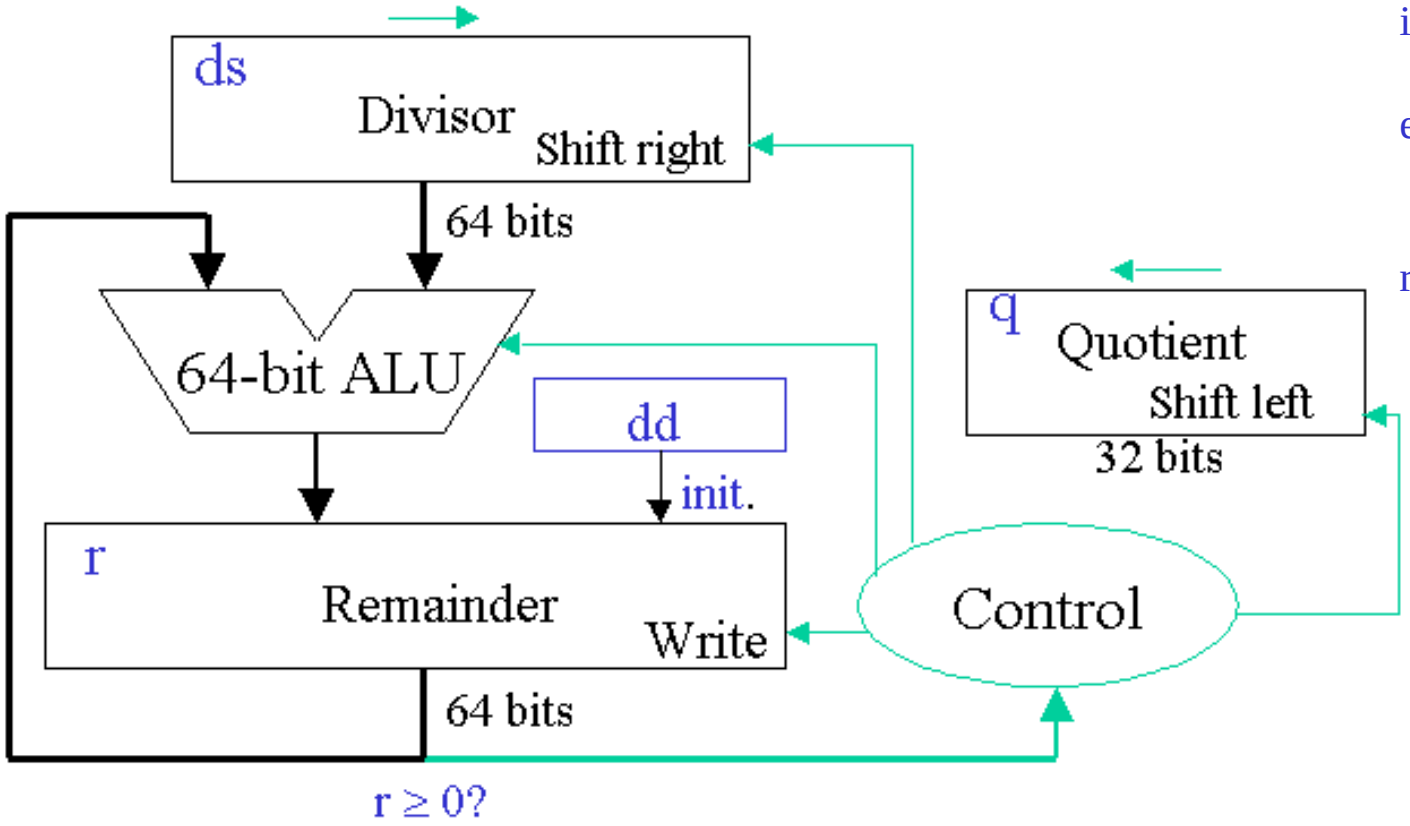
  74 = 9 × 8 + 2

-

```
                1001   q
ds  1000 ) 1001010   dd   (r´ : intermediate value)
           -1000     align MSB(ds) and MSB(dd)
            0010     }  r´< ds: q = q ++ <0>
             0101    }
             1010    ← r´≥ ds: q = q ++ <1>
            -1000
              10     r  (r = r´ when finished)
```

- compare r´ and ds: calculate r´ = r´ - ds
  - r´ < 0, r´ = r´ + ds (restore old value of r´)
  - r´ ≥ 0, accept r´ for further calculation

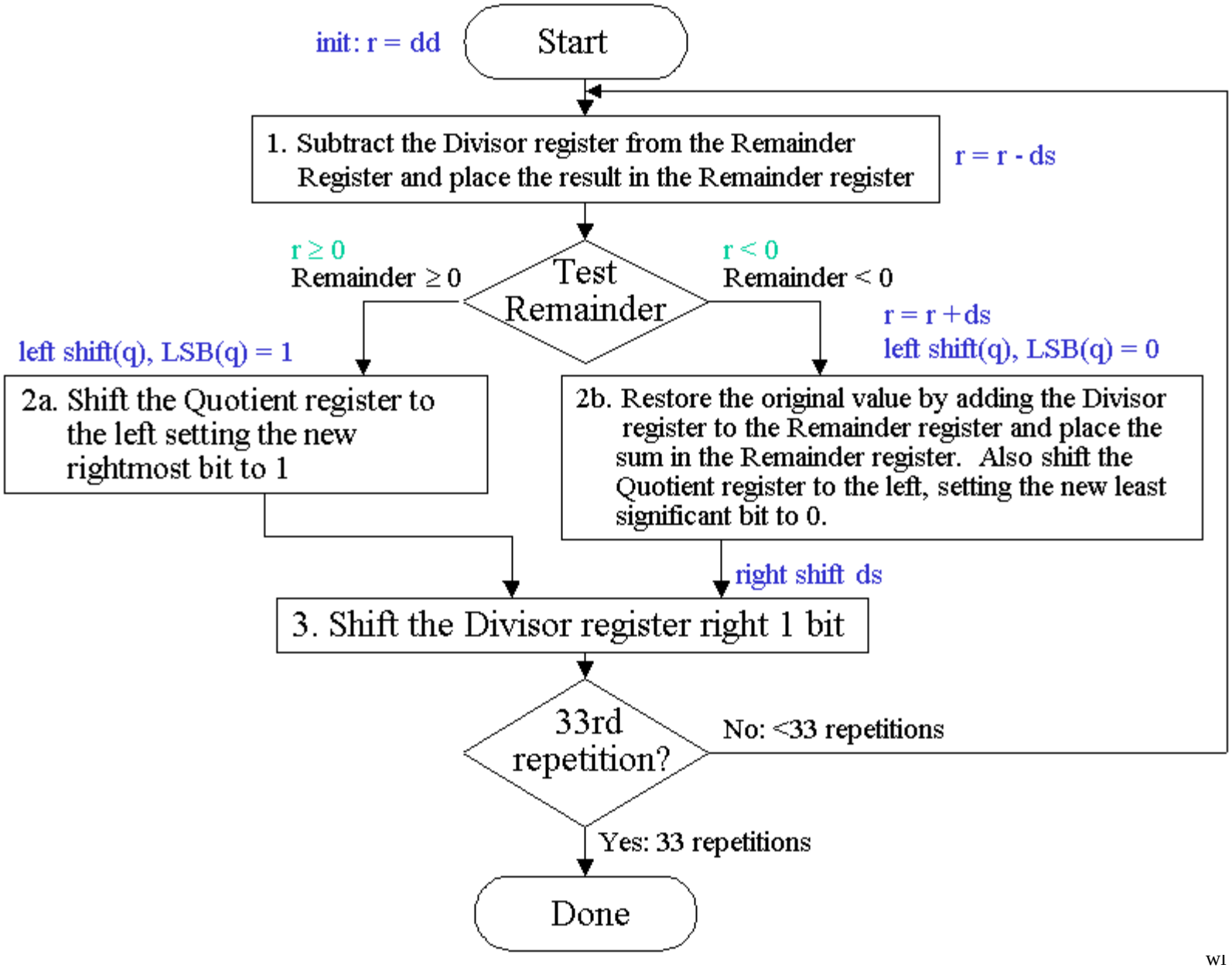
First version of the division hardware



loop: r = r - ds
if r ≥ 0
left shift q, LSB(q)=1
else
r = r + ds, left shift (q)
LSB(q) = 0
right shift ds

The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each step. The remainder is initialised with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

$$dd = (q \times ds) + r$$



Division example using first algorithm

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Values	0000	**0010** 0000	0000 **0111**
1	1: Rem = Rem - Div 2b: Rem<0 ⇒ +Div, sll Q, Q0=0 3: Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	**1**110 0111 0000 0111 0000 0111
2	1: Rem = Rem - Div 2b: Rem<0 ⇒ +Div, sll Q, Q0=0 3: Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	**1**111 0111 0000 0111 0000 0111
3	1: Rem = Rem - Div 2b: Rem<0 ⇒ +Div, sll Q, Q0=0 3: Shift Div right	0000 0000 0000	0000 1000 0000 1000 0000 0100	**1**111 1111 0000 0111 0000 0111
4	1: Rem = Rem - Div 2a: Rem≥0 ⇒ sll Q, Q0=1 3: Shift Div right	0000 0001 0001	0000 0100 0000 0100 0000 0010	**0**000 0011 0000 0011 0000 0011
5	1: Rem = Rem - Div 2a: Rem≥0 ⇒ sll Q, Q0=1 3: Shift Div right	0001 0011 0011	0000 0010 0000 0010 0000 0001	**0**000 0001 0000 0001 0000 0001

dd=7 q=3
ds=2 r=1

To note

- refining division implementation
  - remainder shift left: reduce divisor / ALU size
  - combine quotient and remainder registers
- MIPS instructions
  - multu, divu: unsigned operations
  - result in HI, LO registers
  - mflo: move data from LO register
- exercise: signed numbers in
  - multiplication (3rd Ed: p.180, 4th Ed: p.234, 5th Ed: p.187)
  - division (3rd Ed: p.187, 4th Ed: p.239, 5th Ed: p.193)