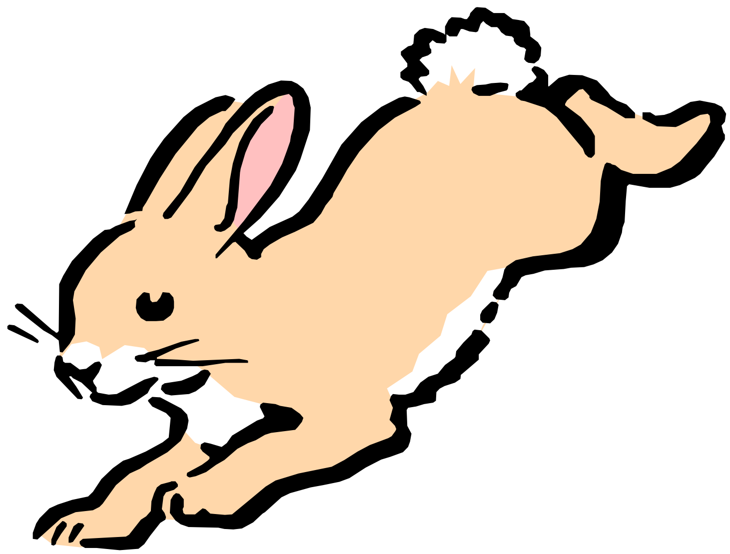
Multiplication

- Multiplication is one of the harder arithmetic operations.
- There are two basic ways to do multiplication in hardware.
  - You can use a lot of hardware and get a relatively fast multiplier.
  - You can use less hardware, but you'll end up with a slower circuit.
- Today we'll see some algorithms for unsigned and signed multiplication.
- These methods are applicable to both hardware and software.



©2001-2003 Howard Huang

Binary multiplication example

- Here is an example of unsigned binary multiplication, for 13 × 6 = 78.

				1	1	0	1	Multiplicand
			×	0	1	1	0	Multiplier
				0	0	0	0	
			1	1	0	1		
		1	1	0	1			Partial products
+	0	0	0	0				
	1	0	0	1	1	1	0	Product

- Since we only multiply by 0 and 1, the partial products are always either 0 or the multiplicand (1101 in this example).
- The partial products must all be added together.
- With two *n*-bit operands, the product has up to 2*n* bits.

Hardware for generating partial products

- One-bit multiplication corresponds to the logical AND operation.

$0 \times 0 = 0 \qquad 0 \times 1 = 0 \qquad 1 \times 0 = 0 \qquad 1 \times 1 = 1$

- This means we can use AND gates to generate each partial product.
  - We can AND each multiplier bit with each multiplicand bit.
  - Each partial product will be either 0 or the multiplicand itself.

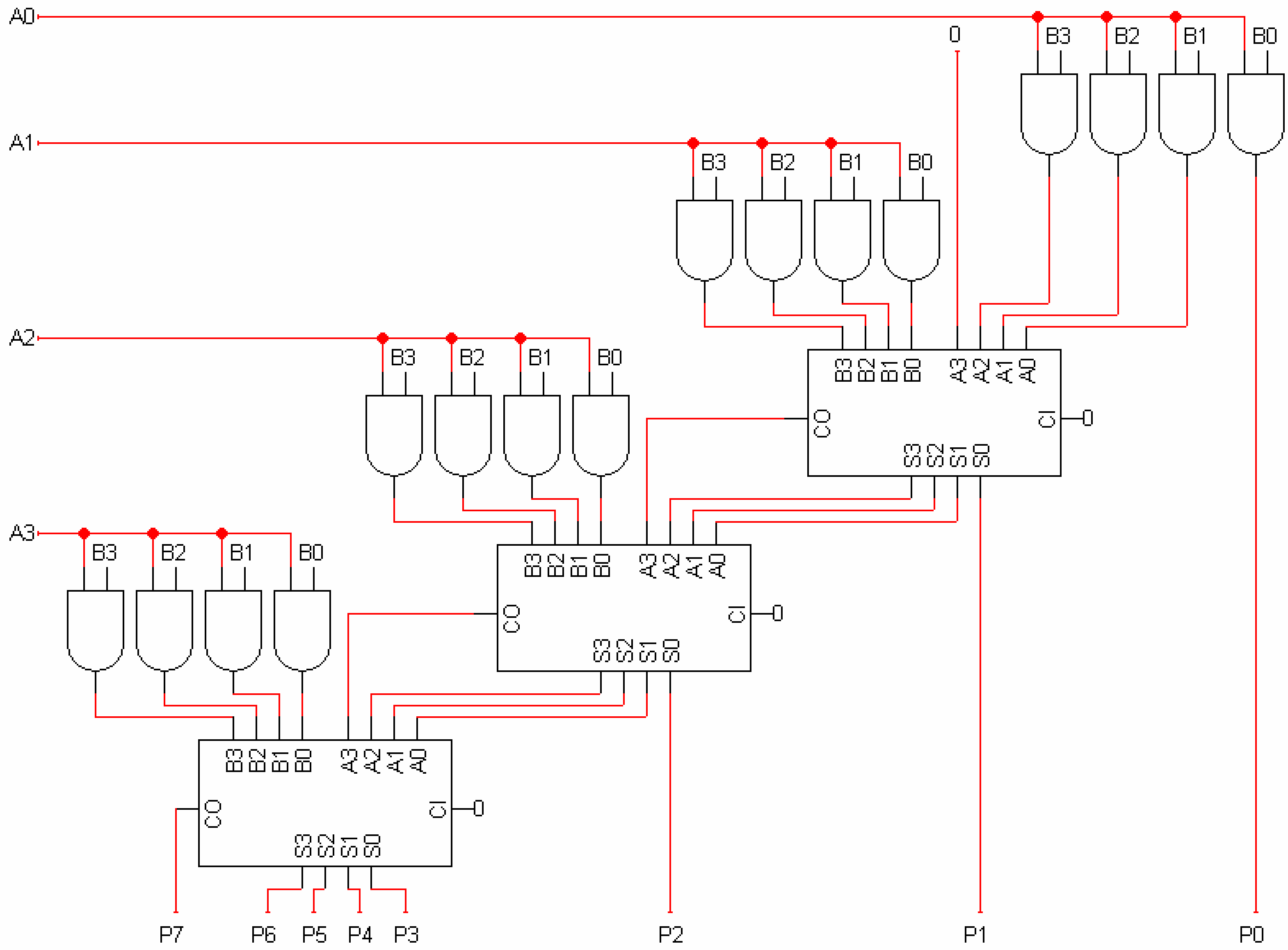
				1	1	0	1	Multiplicand
			×	0	1	1	0	Multiplier
				0	0	0	0	Partial product 0
			1	1	0	1		Partial product 1
		1	1	0	1			Partial product 2
+	0	0	0	0				Partial product 3
	1	0	0	1	1	1	0	Product

Hardware for summing partial products

- How can we add all the partial products together?
  - We have to do $n-1$ additions to sum the n partial products.
  - The final product may have $2n$ bits. But since the partial products are staggered leftwards, we only need to add n bits at a time.
- So we can use $n-1$ adders, each of which adds n bits.

				1	1	0	1	Multiplicand
			×	0	1	1	0	Multiplier
				0	0	0	0	
			1	1	0	1		
		1	1	0	1			Partial products
+	0	0	0	0				
	1	0	0	1	1	1	0	Product

A 4 × 4 multiplier (A × B = P)



Analysis of this approach

- How much hardware is needed to multiply two n-bit numbers?
  - We need n^2 AND gates to generate the n partial products.
  - We use $n-1$ n-bit adders to sum the partial products.
- A circuit for 32-bit multiplication, for example, requires 1,024 AND gates and thirty-one 32-bit adders.
- The biggest hardware and time expense is in all of those adders.
  - Ripple carry adders are slow for large numbers.
  - Other types of adders, like carry lookahead or carry save, will speed things up, but only by introducing even more gates.

Sequential multiplication

- Another idea is to perform the multiplication in several steps, instead of doing it with a combinational circuit.
- Each step could generate and add one partial product.

```
for i = 0 to n–1
    compute partial product i from multiplier bit i
    add the partial product to the final product
```

- We can build this as a sequential circuit.
  - The big advantage is that we'll need just one adder, which is used *n* times in generating the final product. (Compare this with the previous scheme, where *n*–1 adders were each used once.)
  - The disadvantage is that *n* distinct steps are required for the multiply.

Shift registers

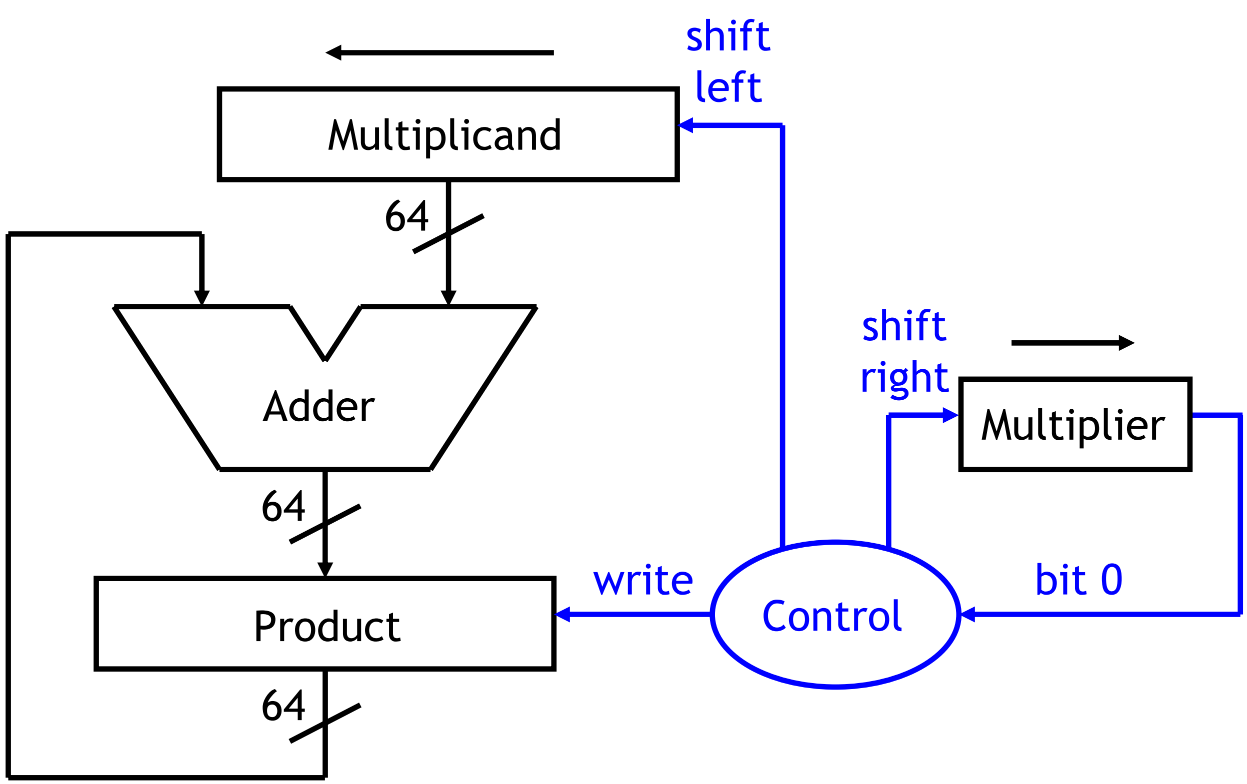
- The product can be $2n$ bits long, so we'll use a $2n$-bit adder for now.
- How can we "stagger" the partial products over successive steps?

					1	1	0	1	Multiplicand
				×	0	1	1	0	Multiplier
	0	0	0	0	0	0	0	0	
	0	0	0	1	1	0	1	0	Partial products
	0	0	1	1	0	1	0	0	
+	0	0	0	0	0	0	0	0	
	0	1	0	0	1	1	1	0	Product

- One solution is to put the multiplicand in a shift register—on each step, we'll shift the multiplicand *left* by one position before ANDing it with the multiplier bits.
- We'll put the multiplier in a shift register too, so it'll be easy to extract each bit with a shift operation.

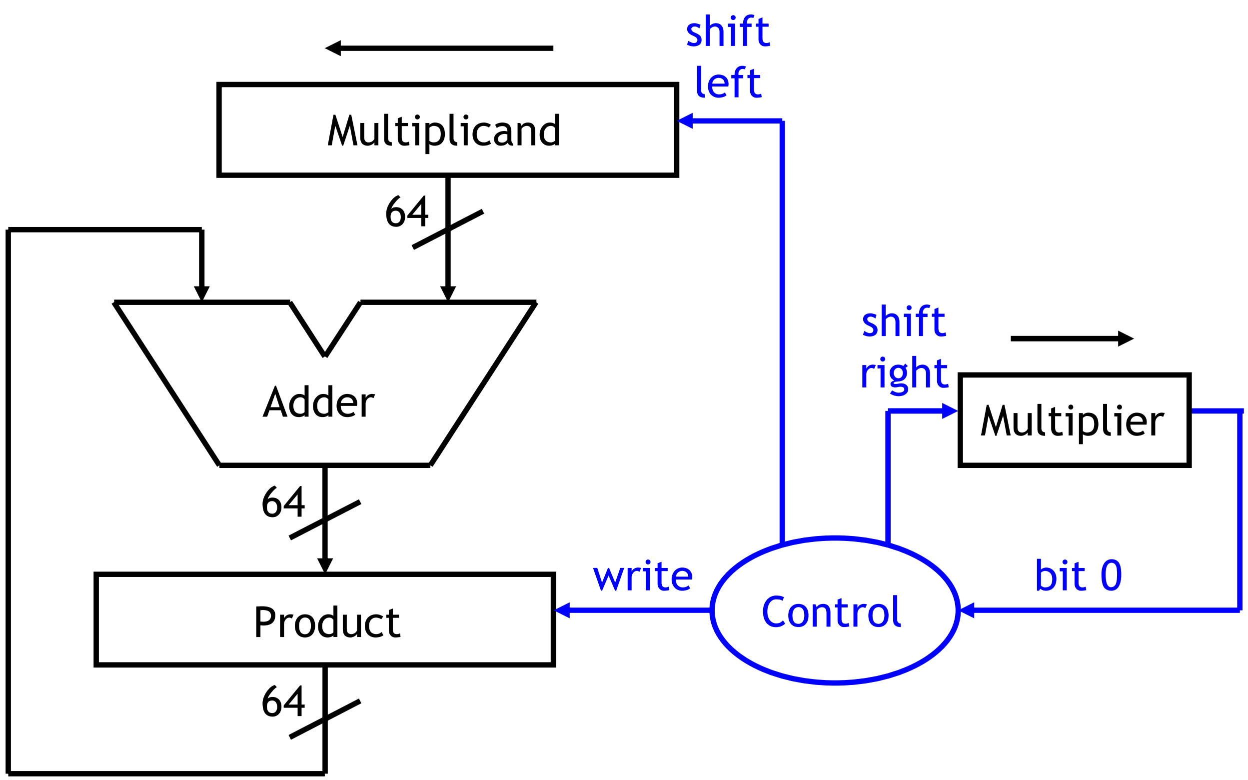
A 32-bit sequential multiplier

- For a 32-bit multiplier, the multiplicand goes into a 64-bit shift register, so we can shift it leftwards and generate all possible partial products.
- The 32-bit multiplier register shifts to the right, so on each step bit 0 of the register will contain the next bit of the multiplier.
- A 64-bit adder sums the current product and the shifted multiplicand.



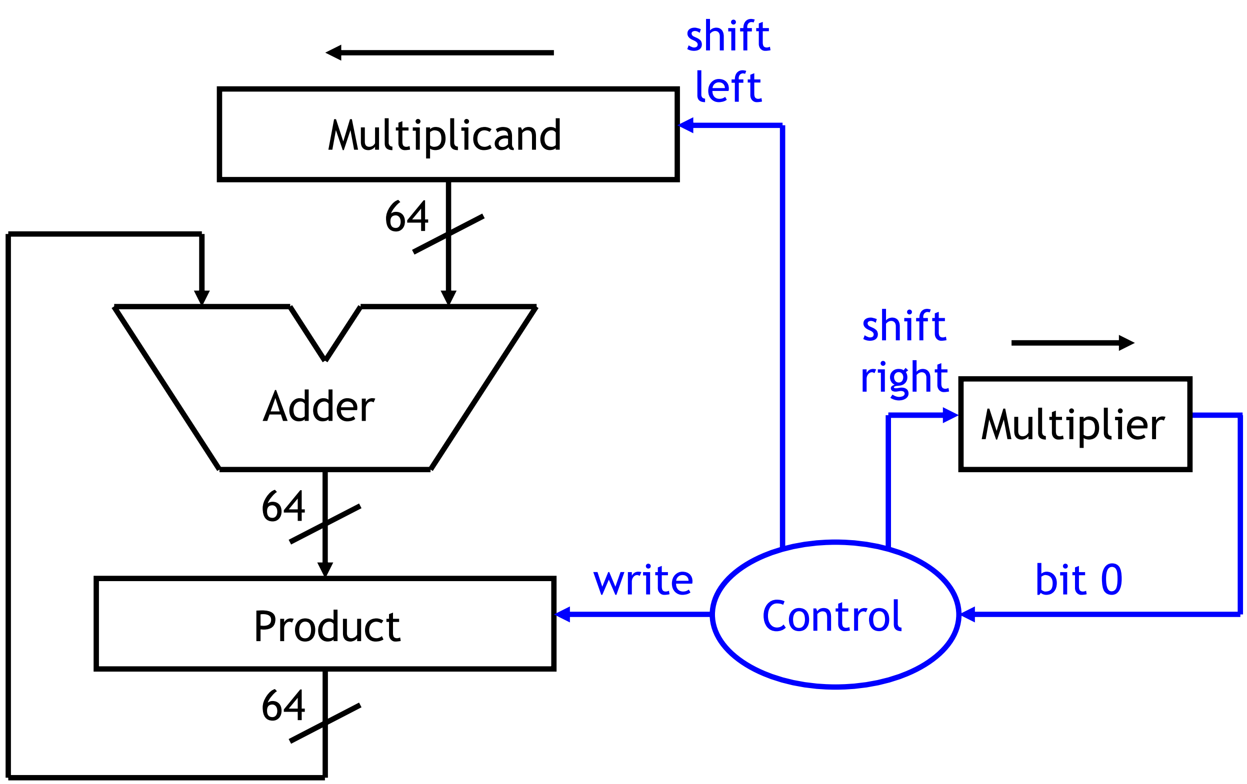
Initialization

- The registers must be initialized before the multiplication can begin.
  - The lower 32 bits of the Multiplicand register should be loaded with the multiplicand, while the upper 32 bits are set to 0.
  - The 32-bit Multiplier register is initialized with the multiplier.
  - The 64-bit product register is cleared to 0.



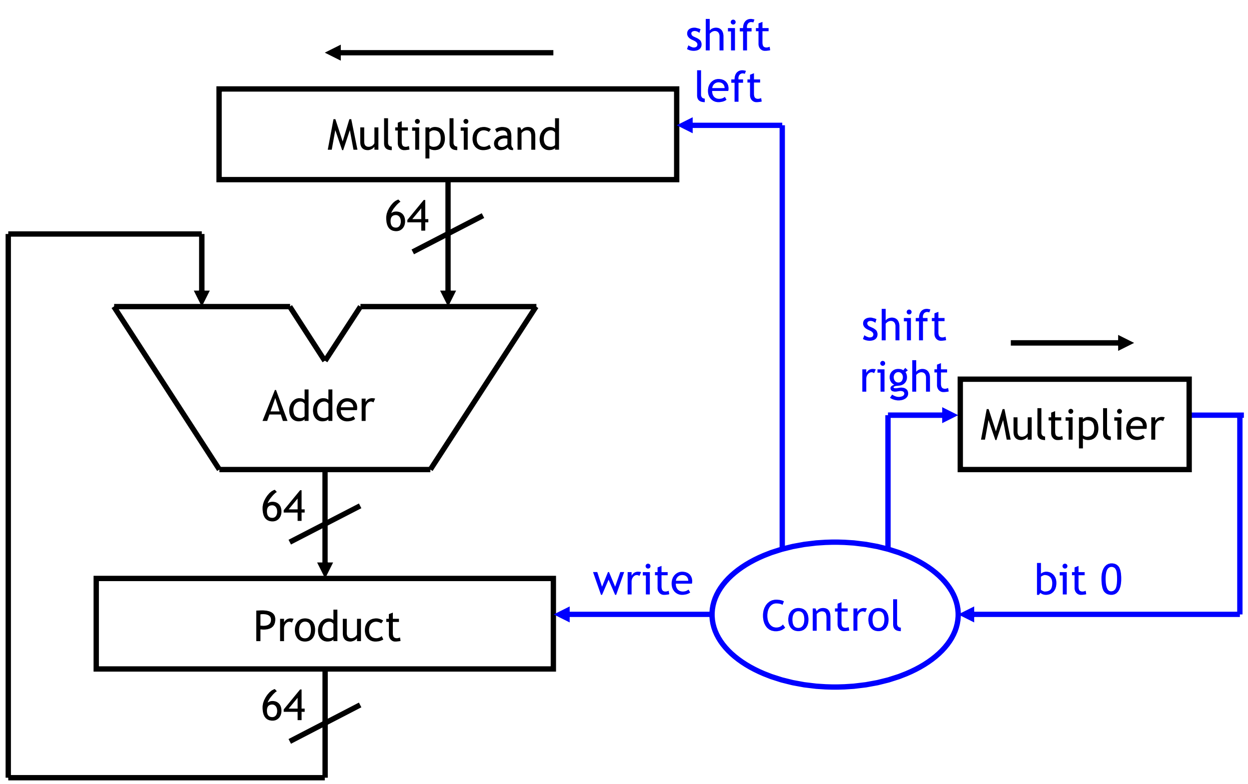
Multiplier control

- In each step, the control unit checks bit 0 of the multiplier register.
  - If the bit is 0 then the corresponding partial product is also 0, so we should leave the Product register alone by setting write=0.
  - If the bit is 1, we have to add the shifted multiplicand to the current Product, so we set write=1.



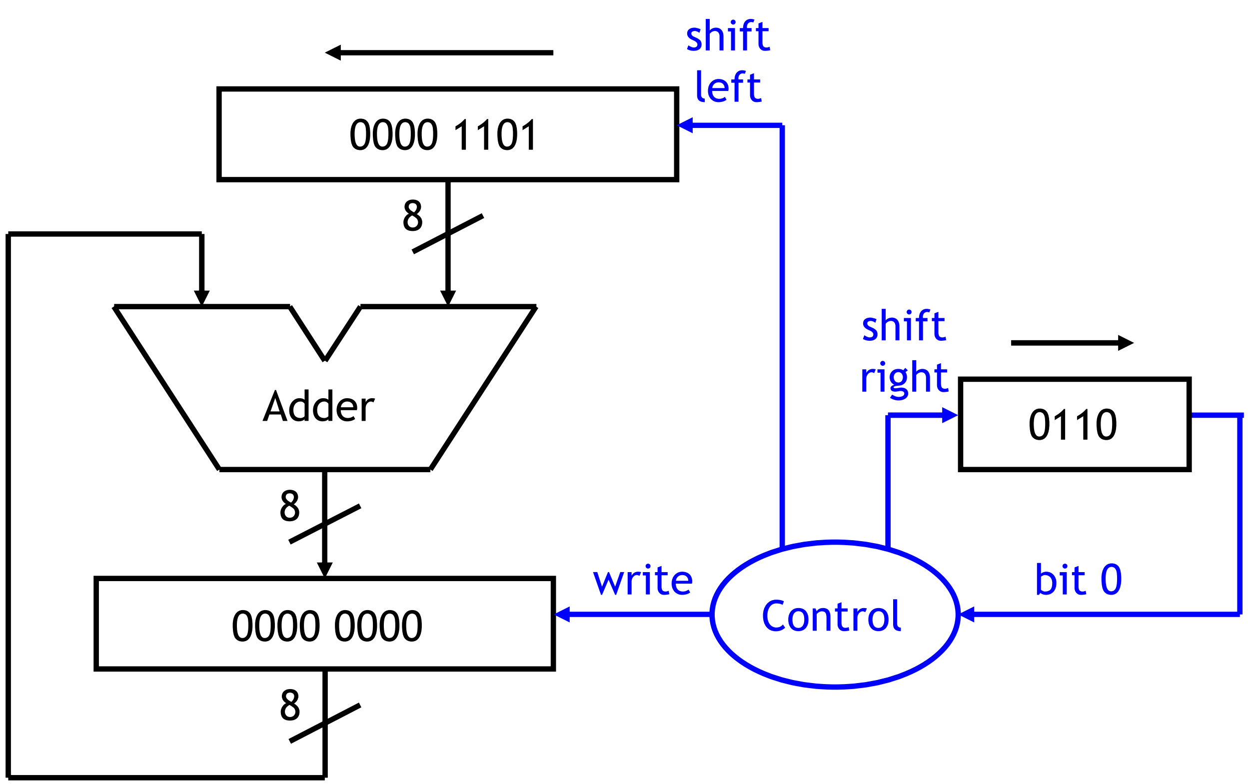
Sequential multiplication algorithm

repeat 32 times
    if bit 0 of Multiplier is 1, then
        write the Adder output to Product
    shift Multiplicand left one bit
    shift Multiplier right one bit



Doing it by hand with 4 bits

- For a four-bit multiplier, we will need one four-bit register, two eight-bit registers and an eight-bit adder.
- Let's multiply 1101 by 0110. The initial register values are shown below.



Step 1a

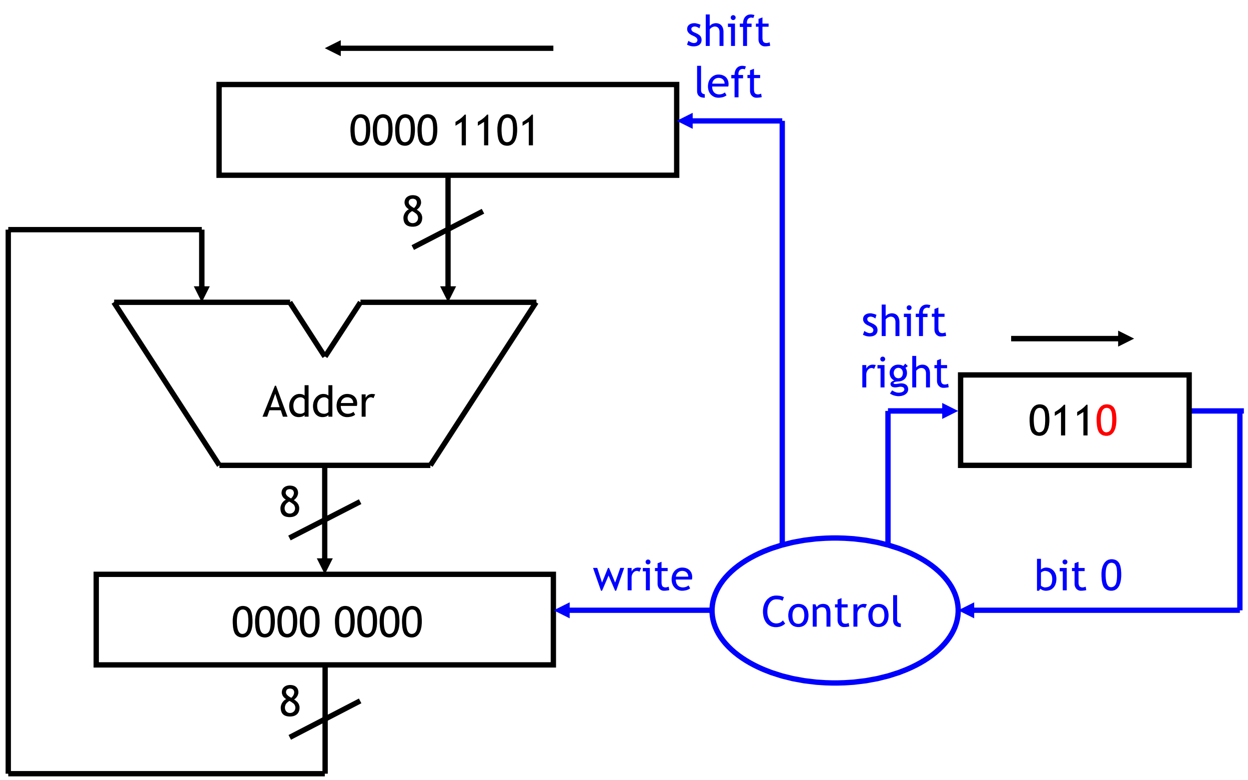
repeat 4 times

if bit 0 of Multiplier is 1, then

write the Adder output to Product

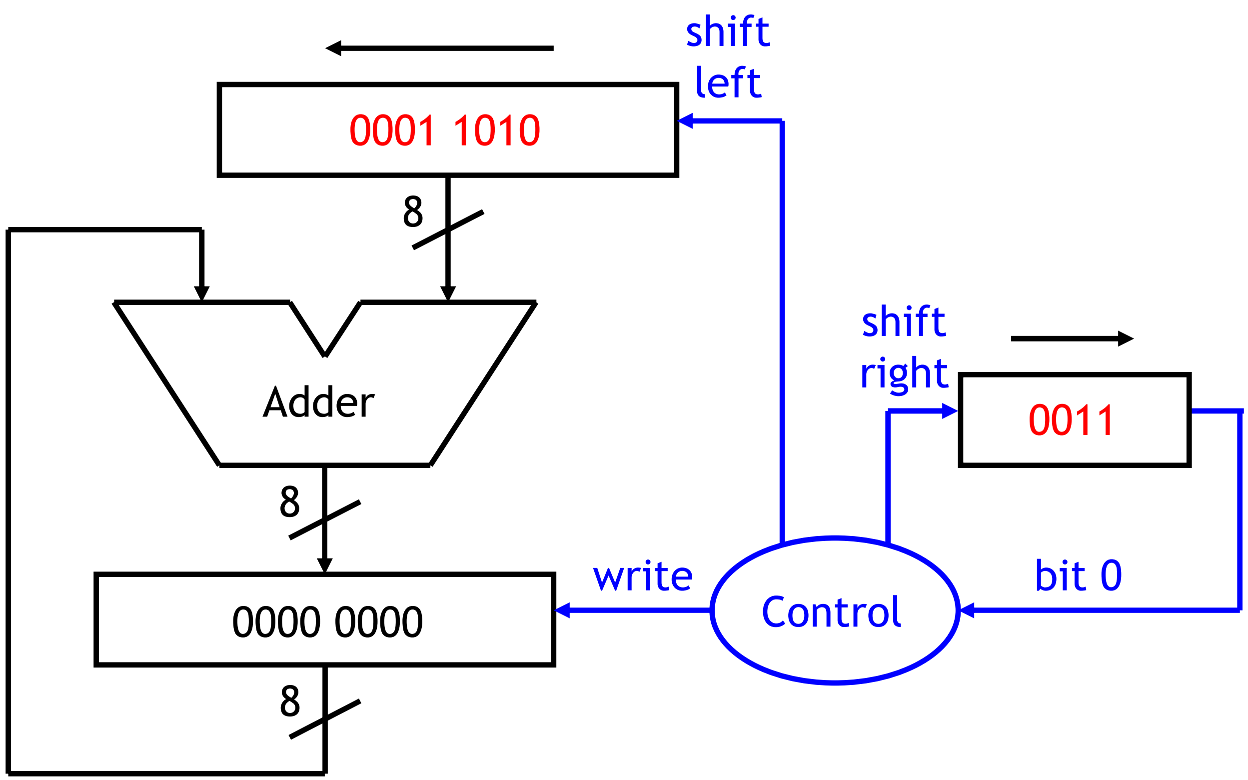
shift Multiplicand left one bit

shift Multiplier right one bit



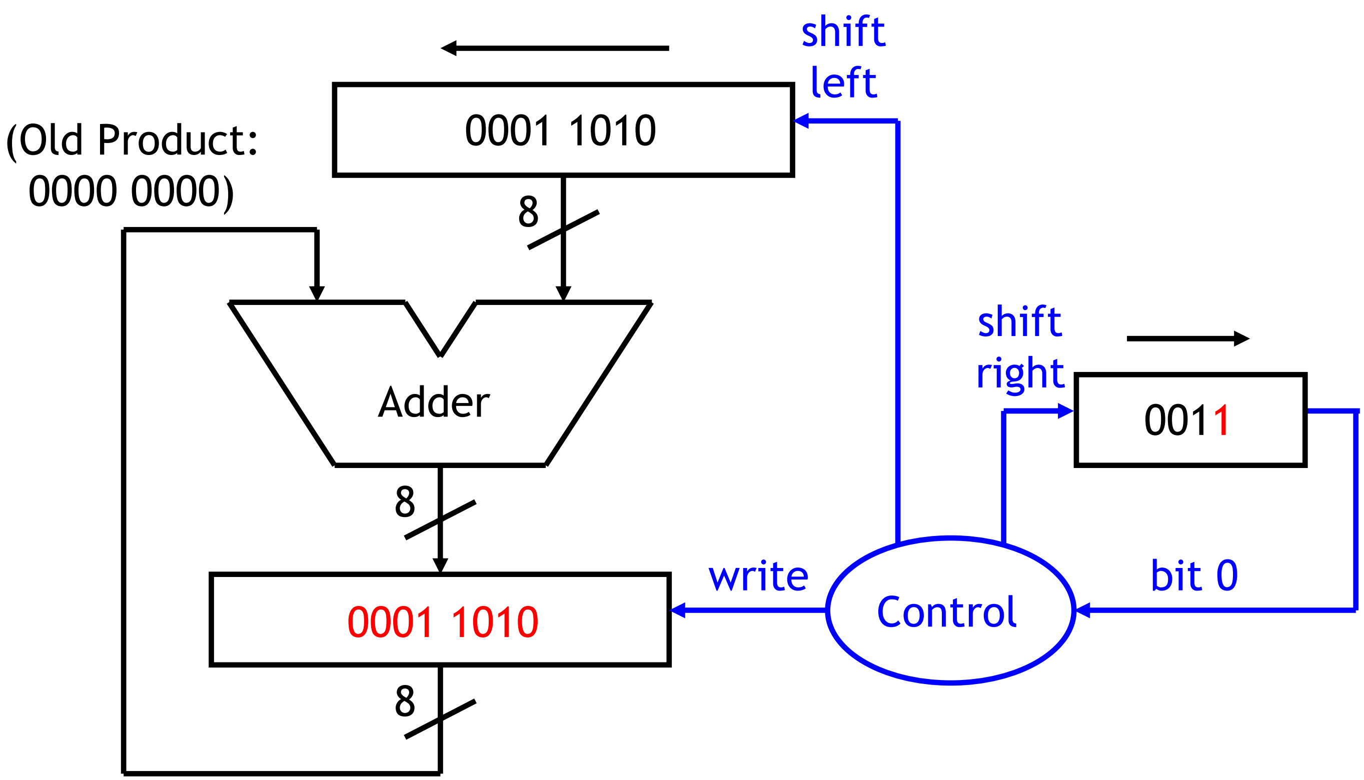
Step 1b

repeat 4 times
    if bit 0 of Multiplier is 1, then
        write the Adder output to Product
    shift Multiplicand left one bit
    shift Multiplier right one bit



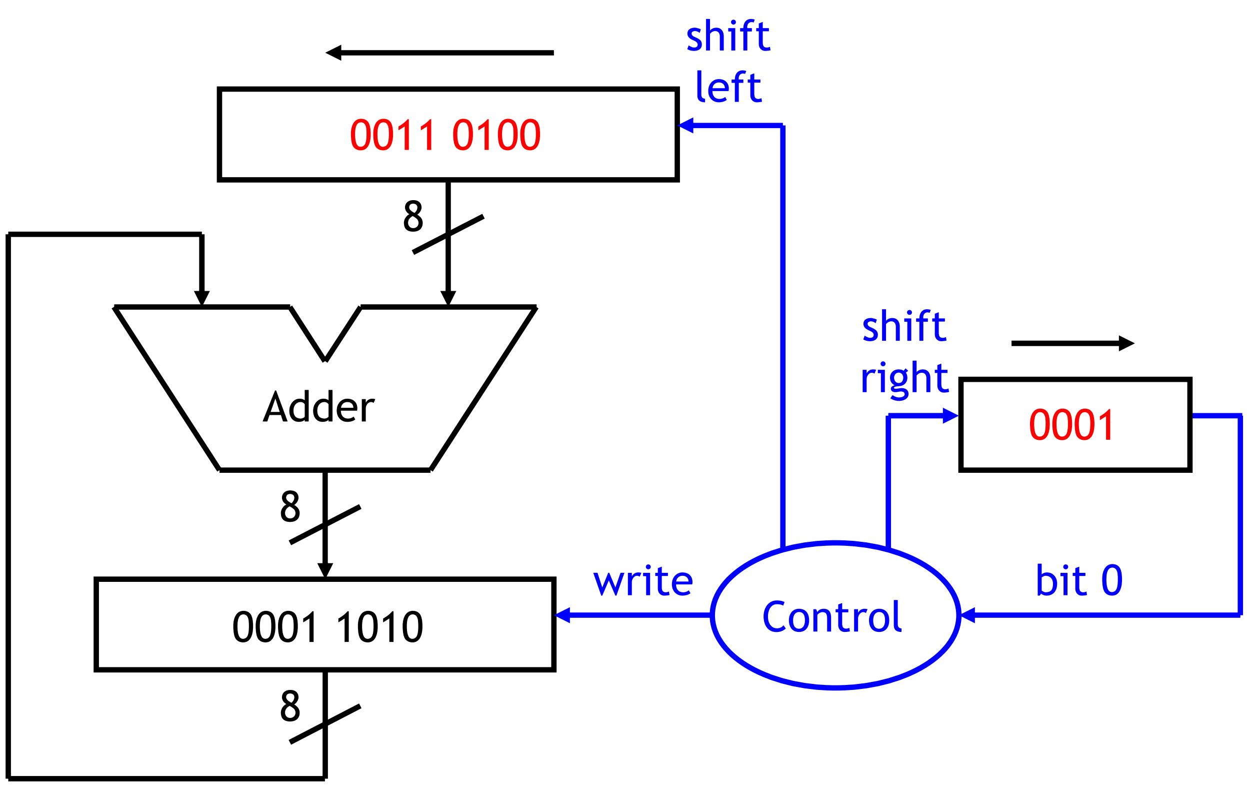
Step 2a

repeat 4 times
- if bit 0 of Multiplier is 1, then
  - write the Adder output to Product
- shift Multiplicand left one bit
- shift Multiplier right one bit



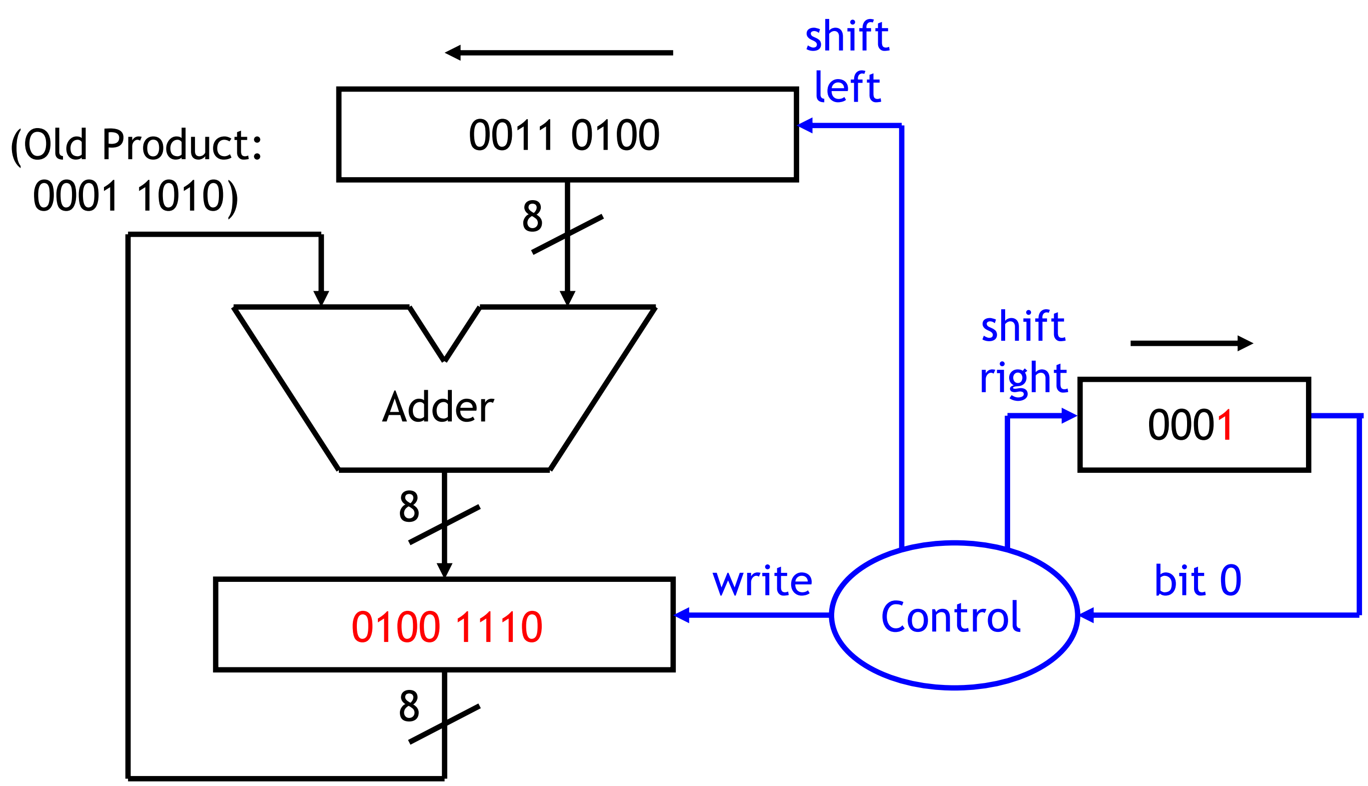
Step 2b

repeat 4 times
    if bit 0 of Multiplier is 1, then
        write the Adder output to Product
    shift Multiplicand left one bit
    shift Multiplier right one bit



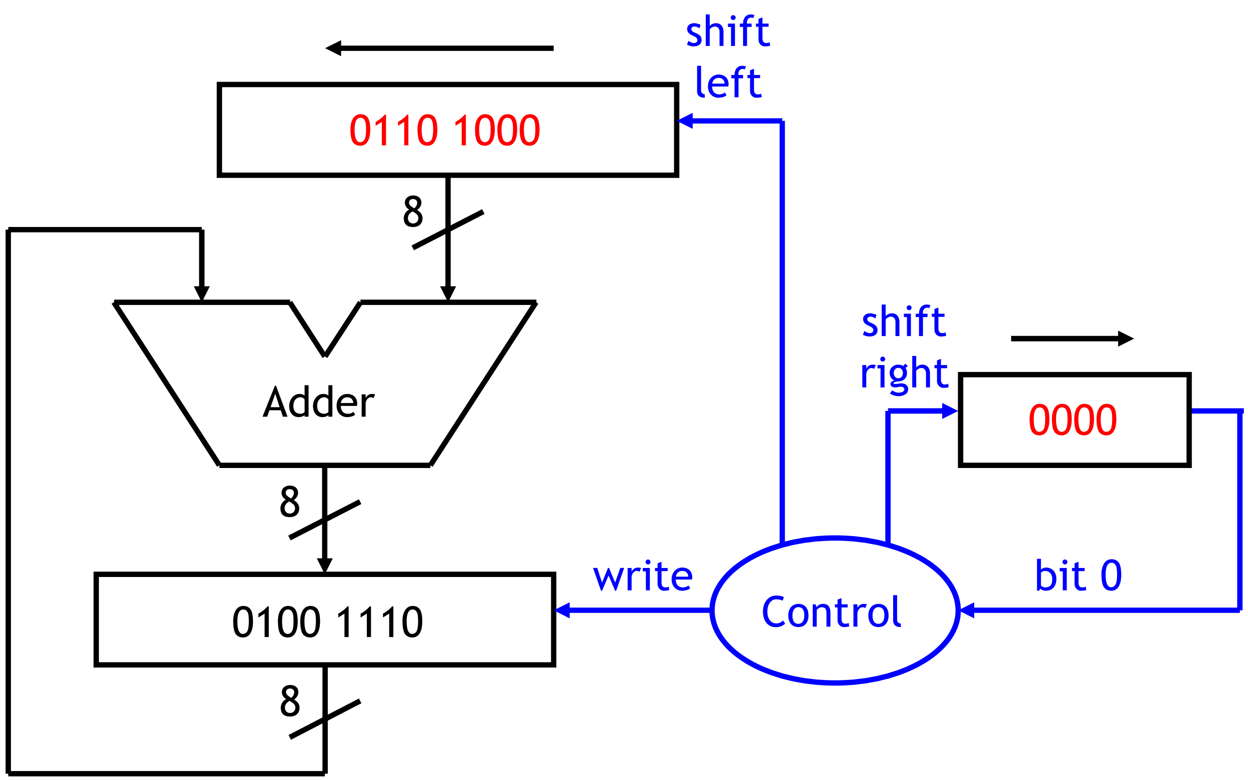
Step 3a

repeat 4 times
    if bit 0 of Multiplier is 1, then
        write the Adder output to Product
    shift Multiplicand left one bit
    shift Multiplier right one bit



Step 3b

repeat 4 times
    if bit 0 of Multiplier is 1, then
        write the Adder output to Product
    shift Multiplicand left one bit
    shift Multiplier right one bit



Step 4a

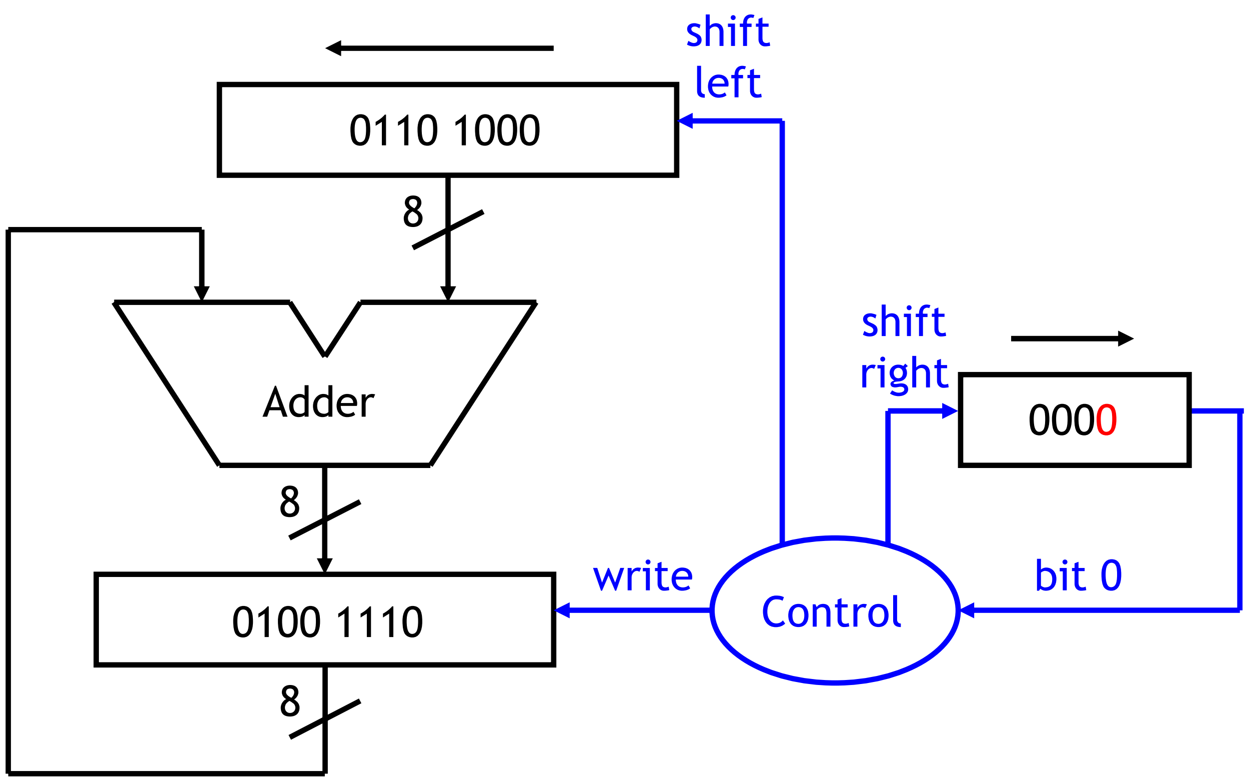
repeat 4 times

if bit 0 of Multiplier is 1, then

write the Adder output to Product

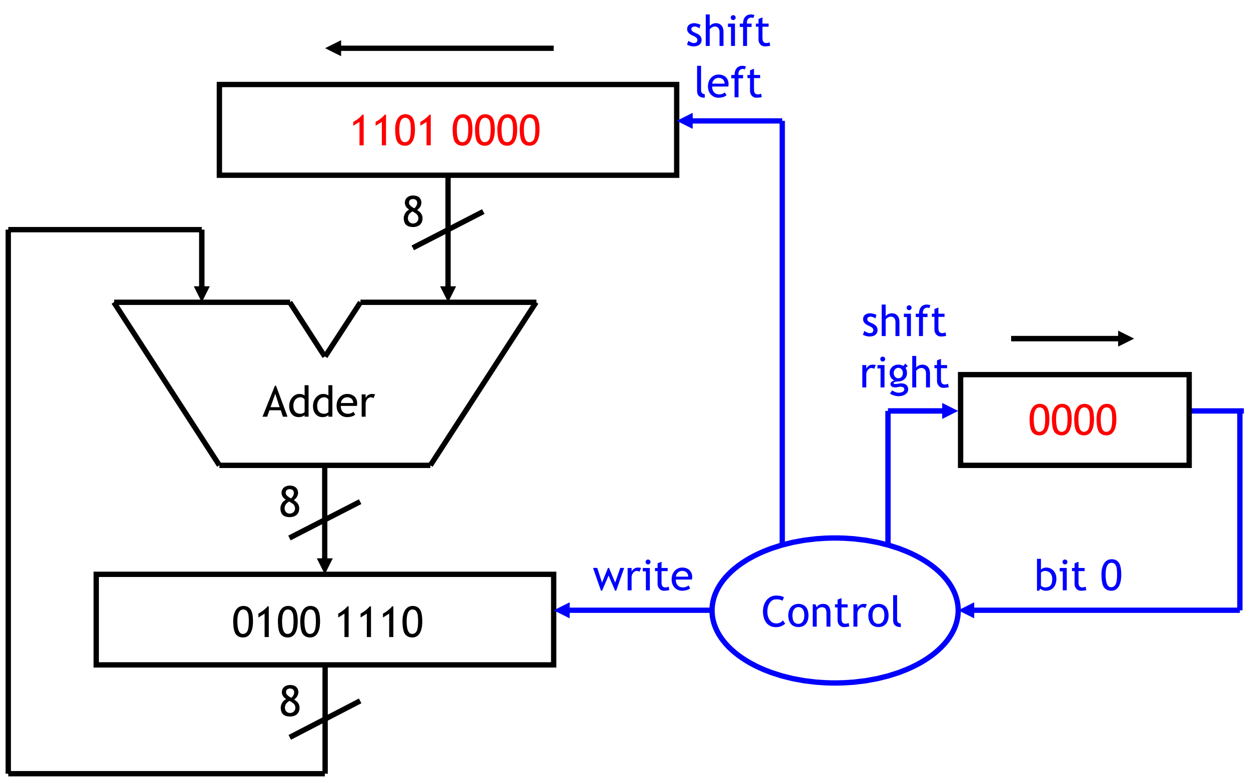
shift Multiplicand left one bit

shift Multiplier right one bit



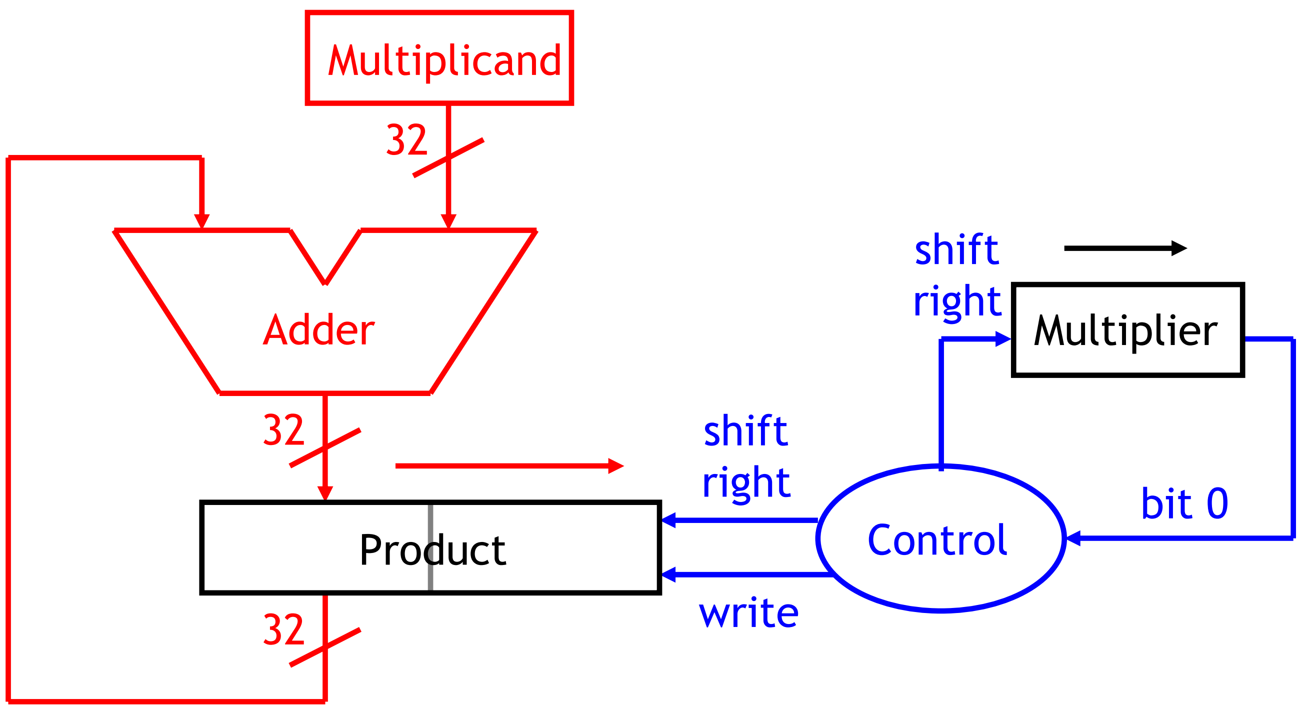
Step 4b (unnecessary)

repeat 4 times
    if bit 0 of Multiplier is 1, then
        write the Adder output to Product
    shift Multiplicand left one bit
    shift Multiplier right one bit



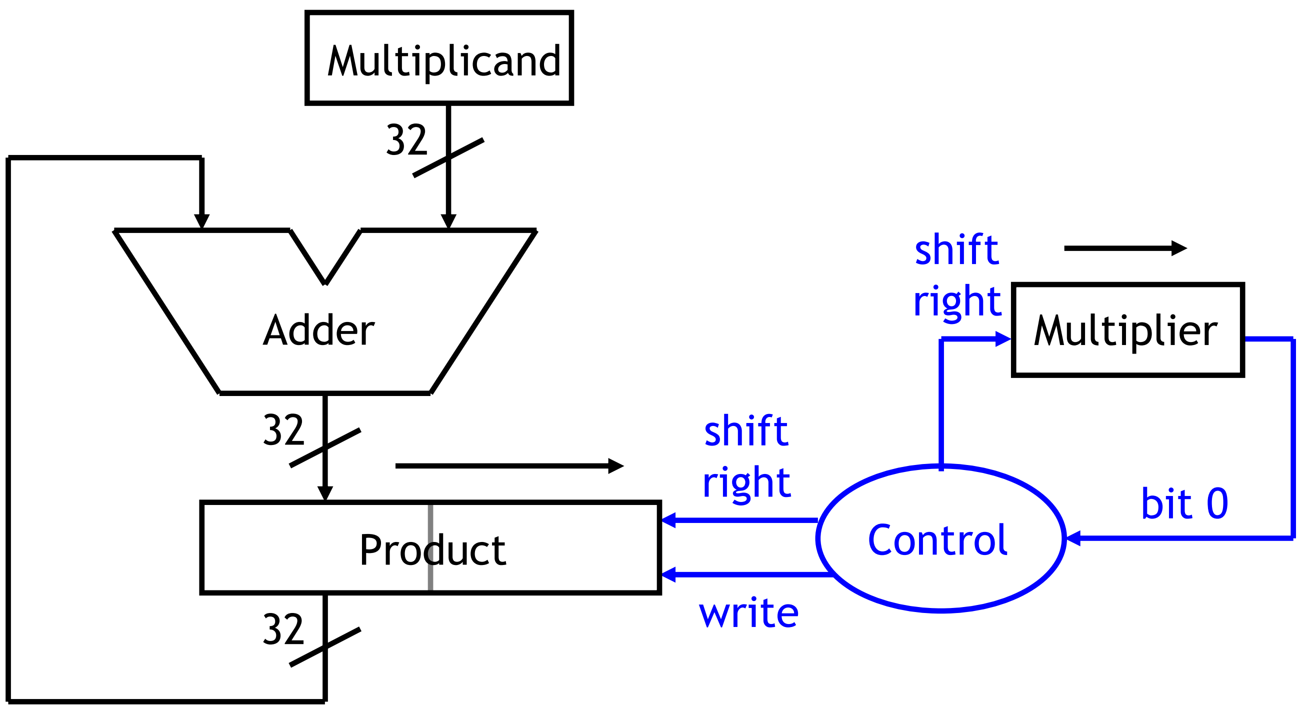
Saving some hardware

- Instead of shifting the multiplicand to the left, we could also shift the *product* to the *right*.
- Using this approach for a 32-bit multiplier saves lots of gates.
  - If we don't shift the multiplicand, we can store it in a 32-bit register.
  - We can also replace the 64-bit adder with a much smaller 32-bit one.



The second sequential multiplier

repeat 32 times
- if bit 0 of Multiplier is 1, then
  - write the Adder output to the left half of Product
- shift Product right one bit
- shift Multiplier right one bit



What about signed multiplication?

- Unfortunately, these circuits don't work for signed numbers.
  - As two's complement numbers, 1101 × 0110 corresponds to –3 × 6.
  - Their product should be 1110 1110 (–18), not 0100 1110 (+78).
- We could use our circuit to multiply the *magnitudes* of two signed values, and then adjust the sign of the result accordingly.
- But this would need extra hardware to test the signs of the operands, and to negate the operands and/or result if necessary.
- Booth's algorithm is a clever sequential multiplication method.
  - It works for signed two's complement numbers.
  - It can also reduce the number of additions needed in a multiplication.

Booth's wonderful idea

- Booth noticed that multiplying x by 2^i–1 is equivalent to multiplying x by 2^i and then subtracting x.

$$(2^i - 1)x = 2^i x - x$$

- This gives us a chance to eliminate some additions in certain cases.
- Consider multiplying 00101 × 00111 (5 × 7).
  - This would normally require three additions, one for each 1 bit in the multiplier.

(00101 × 00001) + (00101 × 00010) + (00101 × 00100)

  - If we did 00101 × 01000 (5 × 8) instead and then subtracted 00101 (5), we would need just one addition and one subtraction.

(00101 × 01000) – (00101 × 00001)

  - In decimal, this corresponds to (5 × 7) = (5 × 8) – (5 × 1).

Generalizing this wonderful idea

- This can be generalized to a sequence of 1s *anywhere* in the multiplier:

$$(2^i - 2^j)x = 2^i x - 2^j x, \text{ where } i > j$$

- Consider 00101 × 01110 (5 × 14), which normally requires three additions.
  - We can rewrite this as (00101 × 10000) – (00101 × 00010).
  - In decimal, $5 \times 14 = (5 \times 16) - (5 \times 2)$.
  - Again, we need just one addition and one subtraction.
- The more consecutive 1s there are in the multiplier, the more addition operations we can eliminate.

Runs of ones

- Booth's algorithm looks for sequences of 1s in the multiplier.
- We need to scan the multiplier *two* bits at a time, from right to left. There are four cases.

Bit i	Bit i-1	Meaning	Example
1	0	Start of a string of 1s	0000111**10**00
1	1	Middle of a string of 1s	000011**11**000
0	1	End of a string of 1s	000**01**111000
0	0	Middle of a string of 0s	00**00**1111000

- The algorithm proceeds by scanning the multiplier bits, two at a time.
  - When a sequence of 1s begins, we'll do a subtraction ($-2^j x$).
  - When a sequence of 1s ends, we'll do an addition ($2^i x$).
- To get things started, we need to add a bit to the right of the original multiplier—this is usually called "bit –1".

Booth's wonderful algorithm

```
initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, preserving the sign
    shift Multiplier right by one position, including bit –1
```

- If we're in the middle of a run of 0s or 1s, then we don't need to do any addition or subtraction—this makes Booth's algorithm potentially faster.
- To make signed multiplication work, the sign of the product has to be *preserved* on right shifts; this is sometimes called an arithmetic shift.

A wonderful example of Booth's wonderful algorithm

- Let's multiply 1101 × 0110 (–3 × 6); the result should be 1110 1110 (–18).

```
initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, preserving the sign
    shift Multiplier right by one position, including bit –1
```

Multiplicand	Product	Multiplier
1101	0000 0000	0110 0

Step 1a

initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, *preserving* the sign
    shift Multiplier right by one position, *including* bit -1

Multiplicand	Product	Multiplier
1101	0000 0000	0110 0

- Since the multiplier bits are 00, we don't need to add or subtract.

Step 1b

initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, *preserving* the sign
    shift Multiplier right by one position, *including* bit –1

Multiplicand	Product	Multiplier
1101	0000 0000	0011 0

Step 2a

initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, *preserving* the sign
    shift Multiplier right by one position, *including* bit –1

Multiplicand	Product	Multiplier
1101	0011 0000	0011 0

- This time the multiplier bits are 10, so we *subtract* Multiplicand (1101) from the left half of Product (originally 0000).

Step 2b

initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, *preserving* the sign
    shift Multiplier right by one position, *including* bit –1

Multiplicand	Product	Multiplier
1101	0001 1000	0001 1

Step 3a

```
initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, preserving the sign
    shift Multiplier right by one position, including bit –1
```

Multiplicand	Product	Multiplier
1101	0001 1000	0001 1

- The multiplier bits are 11, so no adds or subtracts are needed.

Step 3b

initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, *preserving* the sign
    shift Multiplier right by one position, *including* bit –1

Multiplicand	Product	Multiplier
1101	0000 1100	0000 1

Step 4a

```
initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, preserving the sign
    shift Multiplier right by one position, including bit –1
```

Multiplicand	Product	Multiplier
1101	1101 1100	0000 1

- Now the multiplier bits are 01, so we *add* Multiplicand (1101) to the left half of Product (originally 0000).

Step 4b

```
initialize Product to 0, and bit –1 of Multiplier to 0
repeat n times
    if bit 0 and bit –1 of Multiplier are 10, then
        subtract Multiplicand from the left half of Product
    else if bit 0 and bit –1 of Multiplier are 01, then
        add Multiplicand to the left half of Product
    shift Product right by one position, preserving the sign
    shift Multiplier right by one position, including bit –1
```

Multiplicand	Product	Multiplier
1101	1110 1110	0000 0

- The shift right of the Product has to preserve the sign.
- The final result, 1110 1110, is –18 as an 8-bit two's complement number.

MIPS multiplication

- So far we've been using the mul instruction to do multiplications, even though multiplying two 32-bit numbers could yield a 64-bit result.

```
mul   $t0, $t1, $t2
```

- In MIPS, mul is a pseudo-instruction. Multiplication is actually done using mult, which has only two source operands.

```
mult  $t1, $t2
```

- The result goes in two special 32-bit registers called Hi and Lo, which can be copied into regular registers with special one-operand instructions.

```
mfhi  $t3
mflo  $t0
```

- So a MIPS pseudo-instruction like mul $t0, $t1, $t2 is translated into:

```
mult  $t1, $t2
mflo  $t0
```

Summary

- Multiplication is expensive, in terms of both hardware and time.
  - Combinational multipliers need more hardware.
  - Sequential multipliers require more time.
- Booth's algorithm for multiplication has two important advantages.
  - It can handle signed two's complement numbers.
  - It may be able to perform fewer additions.
- The mul instruction in MIPS is really a pseudo-instruction.
  - MIPS uses two special registers Hi and Lo to save the 64-bit result of a 32-bit mult instruction.
  - Special instructions mfhi and mflo are used to access those registers.