CS/COE 0447

Multiplication

wilkie (with content borrowed from:
Jarrett Billingsley
Dr. Bruce Childers)

Multiplication

50 Ways to Move A Number

Negative Re-enforcement?

- Here's where most people would teach you **Booth's Algorithm**
- There are a few problems with it:
  - it's really complicated
  - it's really confusing
  - most importantly, ***literally no one uses it anymore***
    - ***and we haven't for decades***
- As far as I can tell, Booth's Algorithm is a waste of time used to torture architecture students and *nothing more*
  - *Although Booth's Encoding can sometimes be useful*
    - Don't ask me how though because I don't know

Multiplication by repeated addition

- in A × B, the **product** (answer) is "A copies of B, added together"

$$6 \times 3 = 18$$

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18

how many additions would it take to calculate

500,000,000 × 2?

Back to grade school

- Remember your multiplication tables?
- Binary is *so much easier*
- If we list 0 too, the product logic looks awfully familiar…

×	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

×	1
1	1

A	B	P
0	0	0
0	1	0
1	0	0
1	1	1

Just like you remember

- You know how to multiply, riiiight?

```
   0101  =     5  Multiplicand
×  0110  =  ×  6  Multiplier
   ----        --
   0000        30
  01010
 010100
0000000
-------
0011110
```

these are **partial products.** how many **additions** are we doing?

wait, what operation are we doing here...?

Wait, why does this work?

- What are we *actually doing* with this technique?
- remember how positional numbers are really polynomials?

FOIL...

$$78\times54 = 70\times50 + 70\times4 + 8\times50 + 8\times4$$

we're eliminating many addition steps by **grouping them together.**

$$= 78\times50 + 78\times4$$

we group them together by **powers of the base.**

How many bits?

- When we *added* two n-digit/bit numbers, at most how many digits/bits was the sum?
- How about for multiplication?
- When you **multiply** two n-digit/bit numbers, the product will be at most **2n** digits/bits
- So if we multiply two 32-bit numbers…
  - we could get a **64-bit result!** AAAA!
  - if we just ignored those extra 32 bits, or crashed, we'd be losing a lot of info.
  - so we have to **store it.**

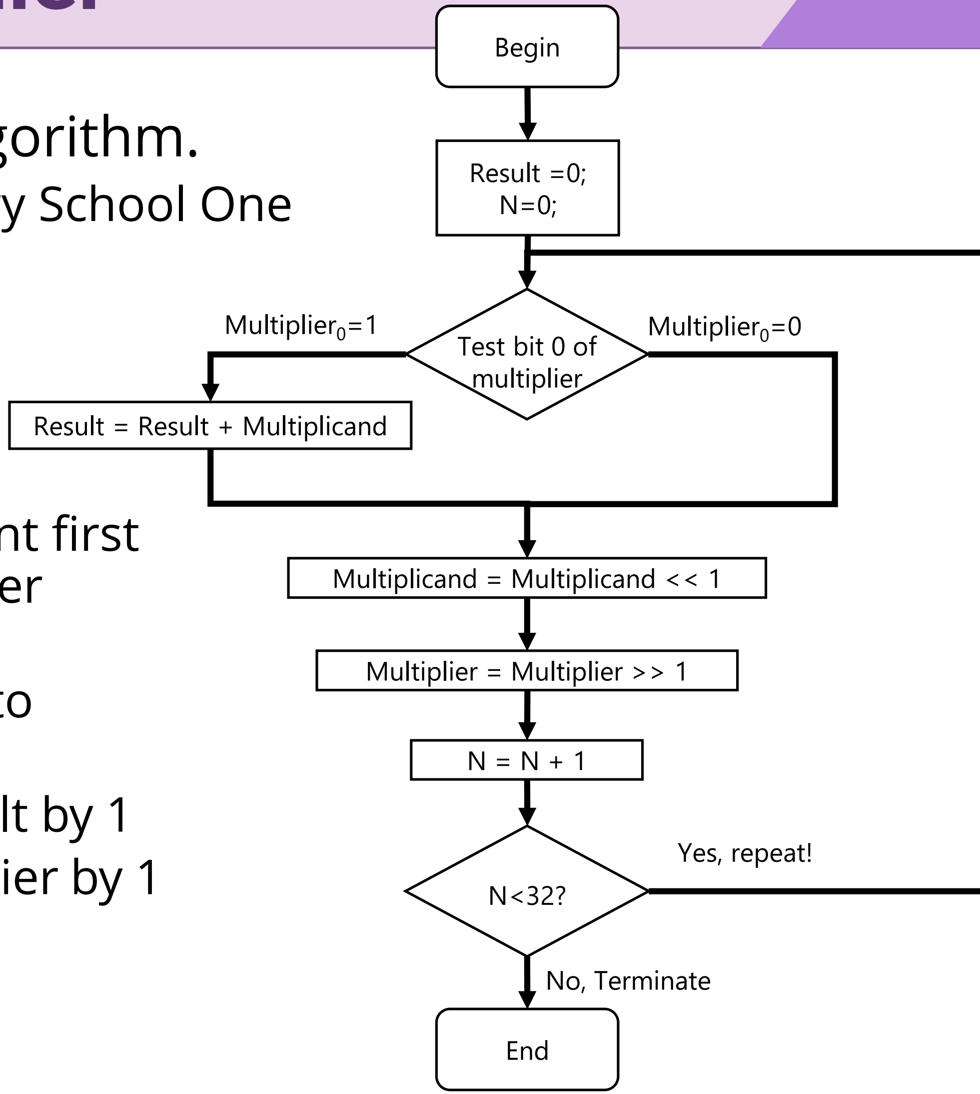
```
  99
× 99
9801
```

```
    9999
×   9999
99980001
```

```
   1111
×  1111
1110001
```

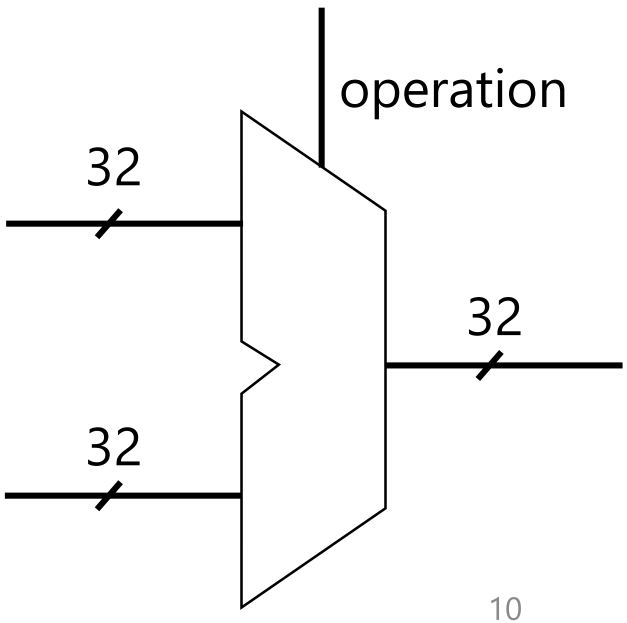
32-bit multiplier

- Here's the first algorithm.
  - It's the Elementary School One
- For each bit in our multiplier,
  - Look at the current first bit of the multiplier
  - If it is a "1", add the multiplicand to the result
  - Shift left our result by 1
  - Shift right multiplier by 1

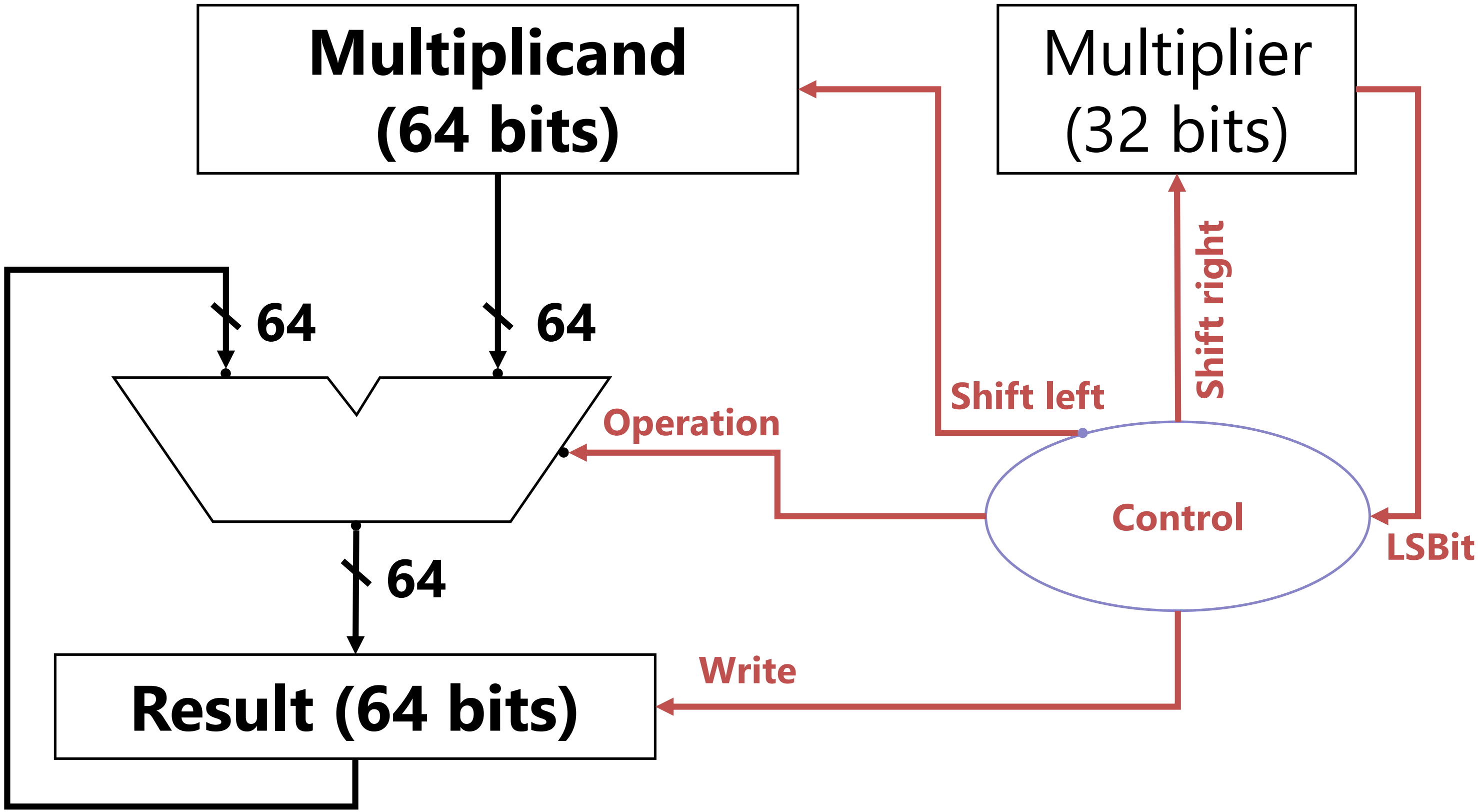


What is this O.o

- It’s an ALU!
  - Arithmetic and Logic Unit
- What does it do?
  - It adds and subtracts numbers
  - It also does logical operations
    OR, AND, NOR, NAND, etc.
- What about overflows?
  - Let’s not worry about those right away ☺
- But how does it work??
  - Soon…



Hardware multiplier – Version 1



In version 1 of the multiplier, the **ALU** and the registers **Multiplicand** and **Result** are 64-bit registers

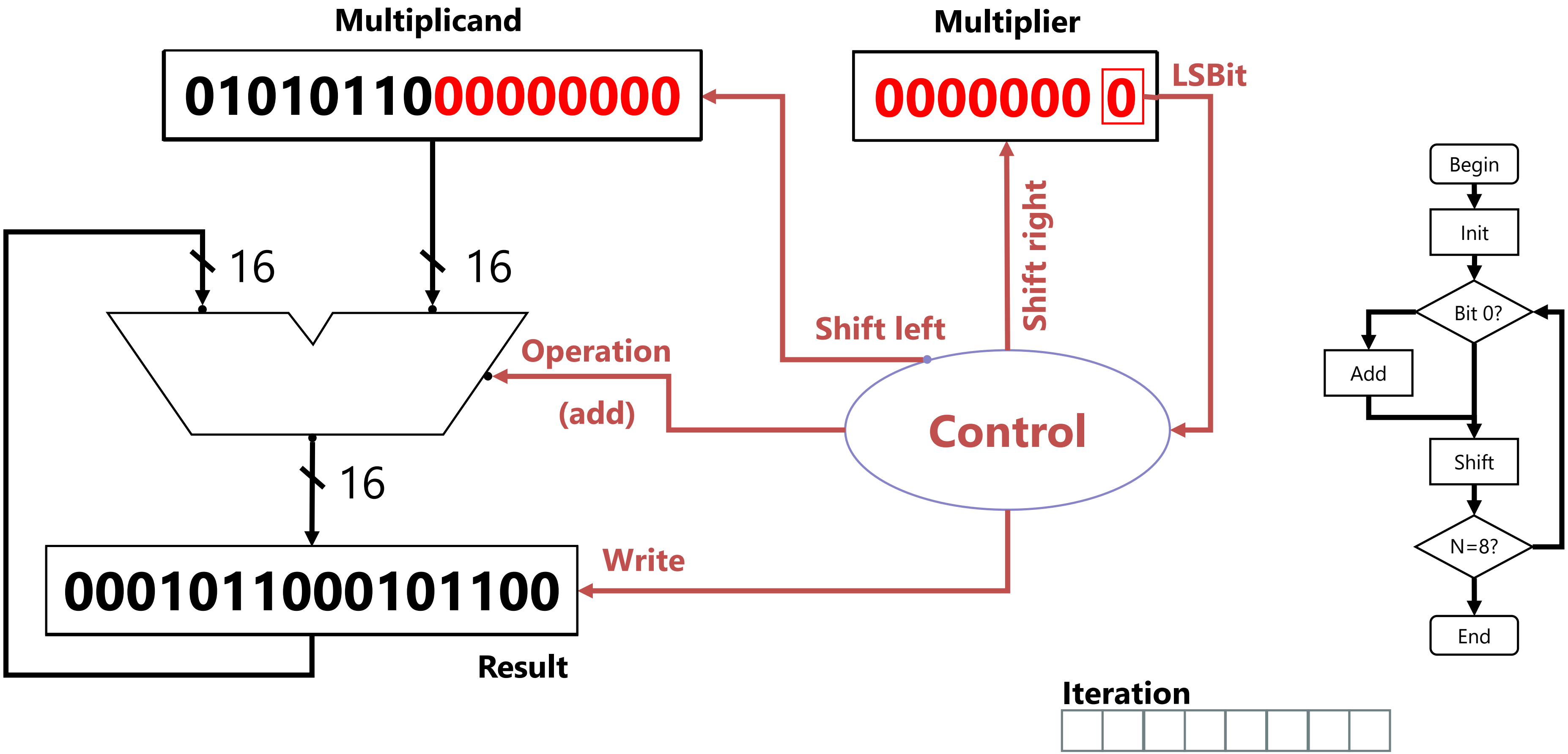
Example 1

- Let’s exemplify by multiplying 86 by 66. The result should be 5676.

```
        01010110
      × 01000010
        00000000
    +  01010110
       010101100
+ 01010110
 001011000101100
```

Hardware multiplier – Version 1

(example with 8 bit registers: 01000010 x 01010110 = 00010110 00101100)



Let's think about this

- There is a relative movement between the Multiplicand and the Result!
  - The shift
  - What if we moved the result to the right instead?

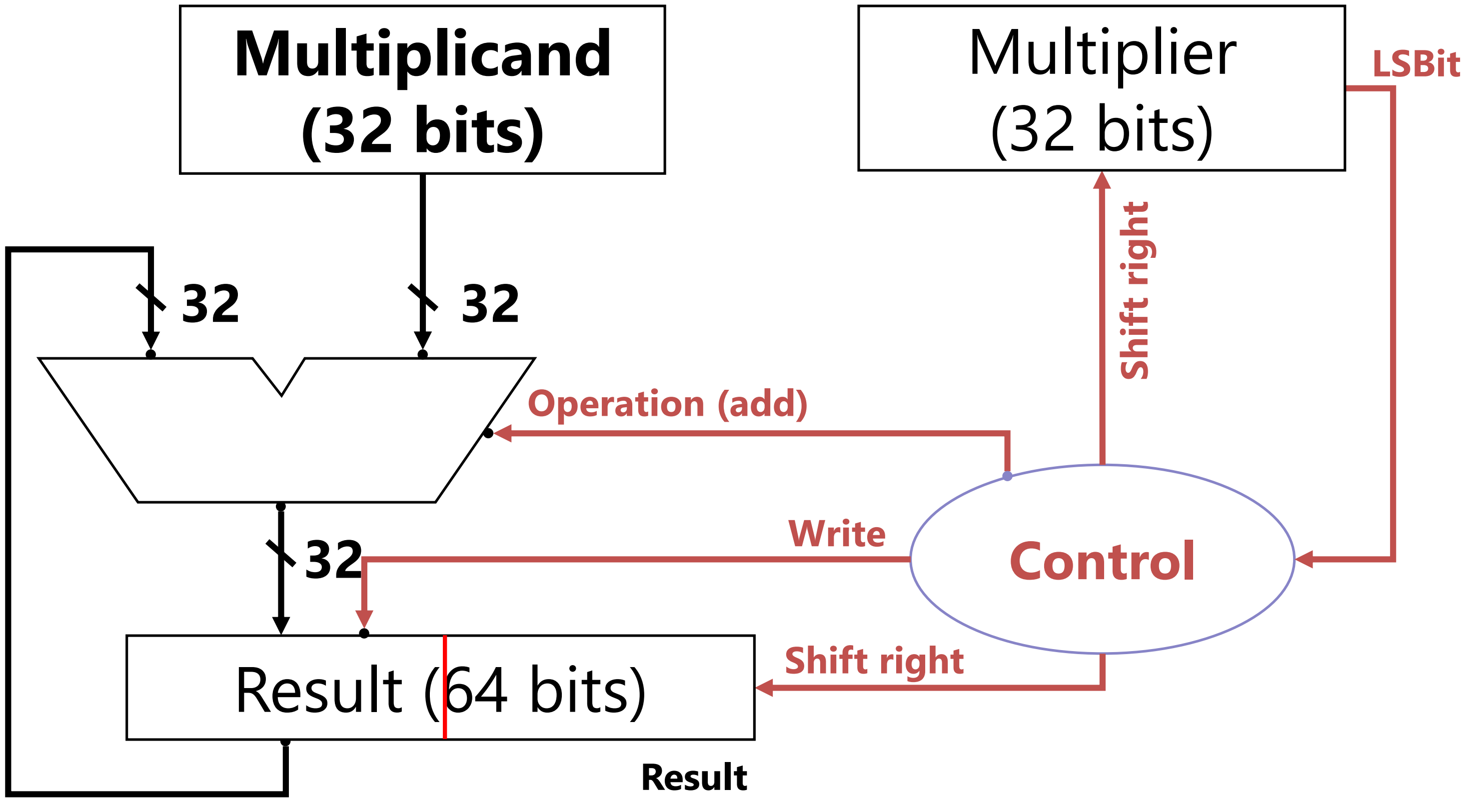
```
     0000
+    0000
     0000
+   0101
    01010
+  0101
   011110
+ 0000
  0011110
```

```
  0000
+ 0000
  0000
+ 0101
  01010
+ 0101
  011110
+ 0000
  0011110
```

Let’s think about this

- We also see that in every iteration, the value of a bit is set to it’s final value.
  - Starting in the LSB moving to the MSB
- And the top bit is always 0
  - So there is no carry!
  - 0+0=0 and 0+1=1
- So we are only operating on 32 bits
  - Therefore we only need a 32-bit adder

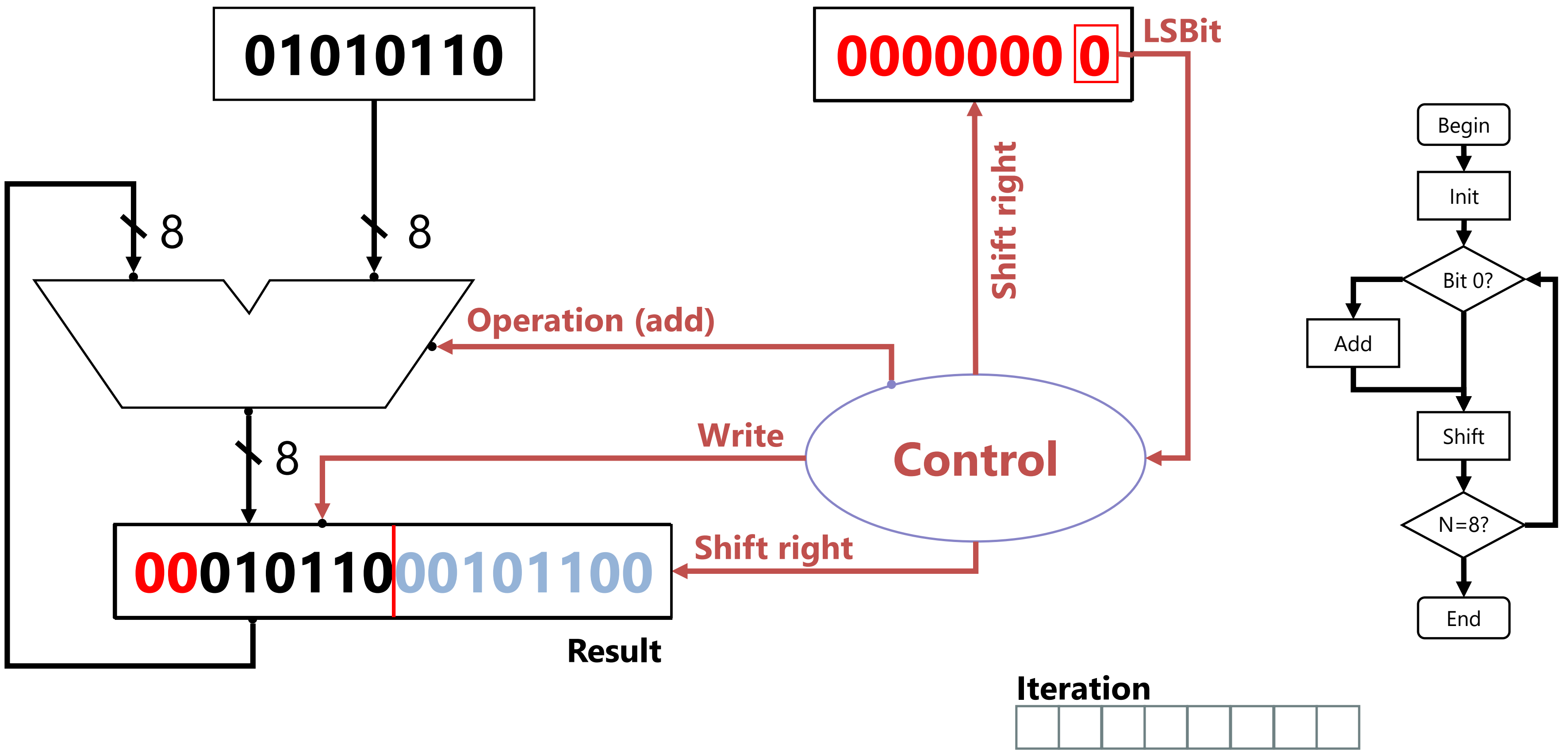
Hardware multiplier – Version 2



In version 2 of the multiplier, the **ALU** and the **Multiplicand** register only need to be 32-bit registers

Hardware multiplier – Version 2

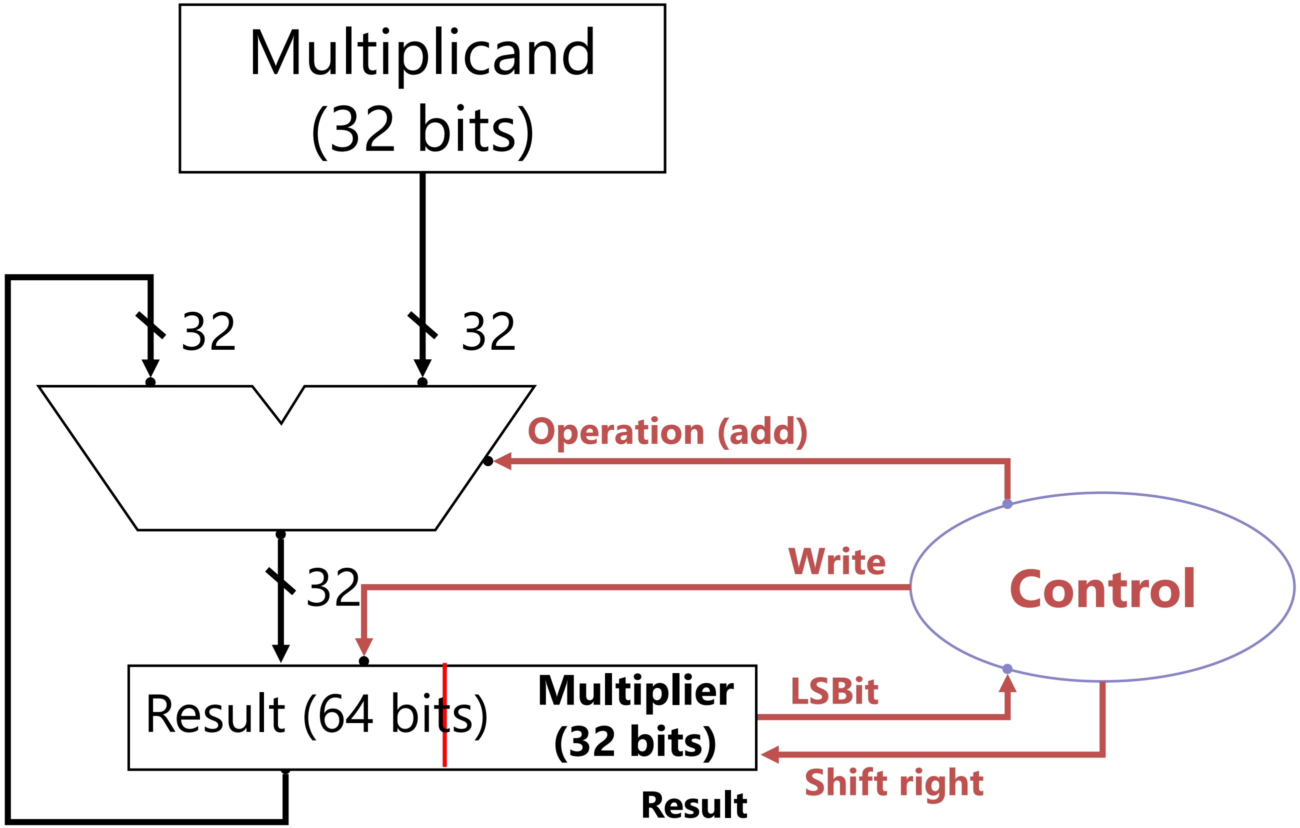
(example with 8 bit registers: 01000010 x 01010110 = 00010110 00101100)



Let's think about this

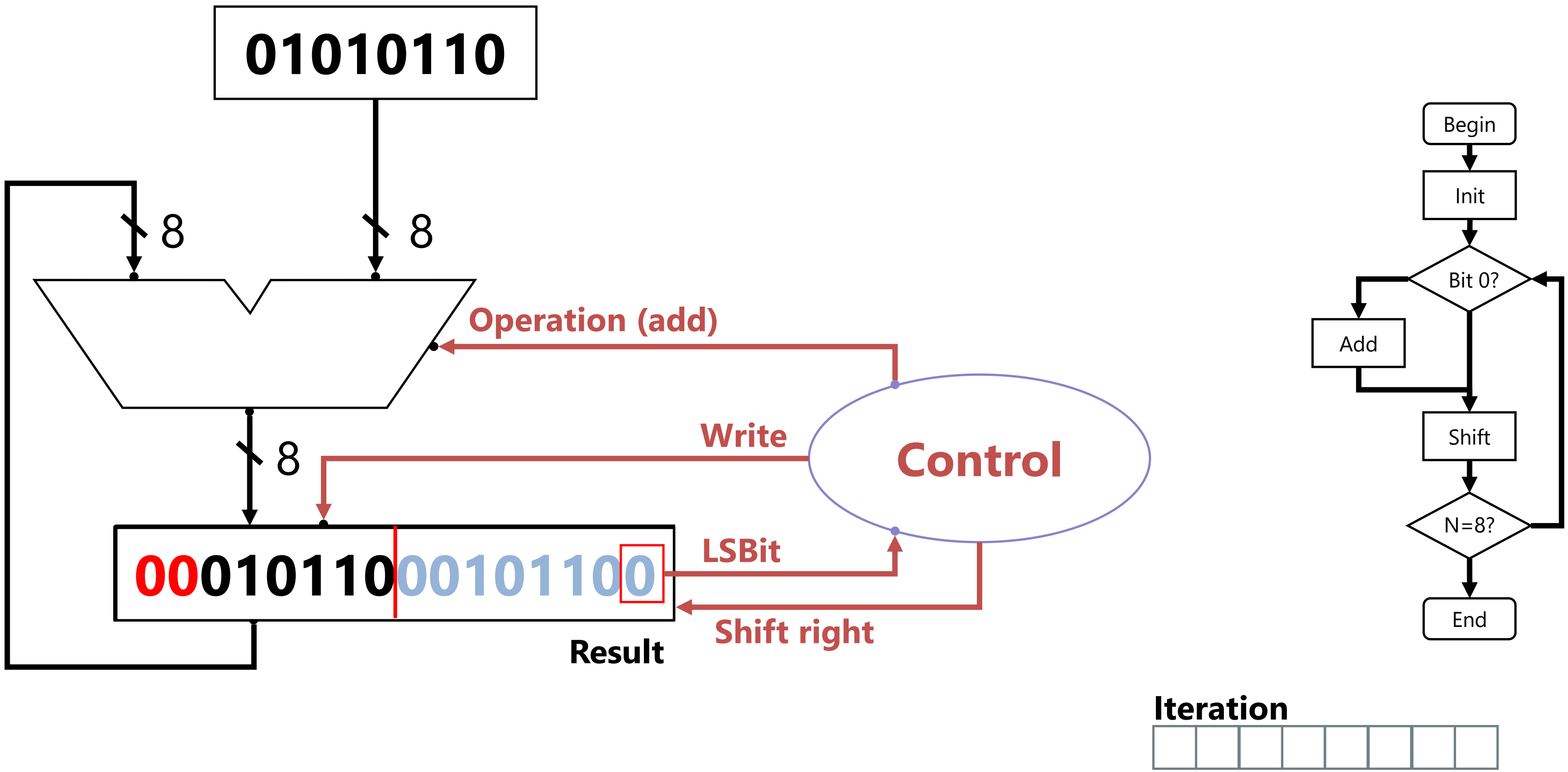
- Both the Result and the Multiplier register are shifted right identically.
- Every time we add a new bit to the result, we lose a bit in the Multiplier
- What if we store the Multiplier on the less significant part of the Result register?
  - This would reduce the amount of space required by multiplication

Hardware multiplier – Version 3



In version 3 of the multiplier, the **Multiplier** register is removed, its value is stored in the least significant half of the result.

Hardware multiplier – Version 3



How (and why) MIPS does it

- MIPS has **two more 32-bit registers, HI** and **LO.** if you do this:

```
mult t0, a0
```

- then HI = **upper 32 bits** of the product and LO = **lower 32 bits**
- to actually get the product, we use these:

```
mfhi t0 # move From HI (t0 = HI)
mflo t1 # move From LO (t1 = LO)
```

- the **mul** pseudo-op does a **mult** followed by an **mflo**
- MIPS does this for 2 reasons:
  - multiplication can *take longer than addition*
  - we'd otherwise have to *change two different registers at once*
- if you wanted to check for 32-bit multiplication overflow, how could you do it?

Signed multiplication

Multiplication can be mean and negative, too

Grade school (but like, 5th, instead of 3rd)

- if you multiply two **signed** numbers, what's the rule?

Product

A	B	P
3	5	**15**
3	-5	**-15**
-3	5	**-15**
-3	-5	**15**

Sign

A	B	S
+	**+**	**+**
+	**-**	**-**
-	**+**	**-**
-	**-**	**+**

if the signs of the operands **differ**, the output is **negative**.

Don't repeat yourself

- We already have an algorithm to multiply **unsigned** numbers
- Multiplying signed numbers is exactly the same (except for the signs)
- So why not use what we already made?

```
long prod = unsigned_mult(abs(A), abs(B));
if(sgn(A) == sgn(B))
    return prod;
else
    return -prod;
```