ECE/CS 552: Integer Multipliers

© Prof. Mikko Lipasti

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

Basic Arithmetic and the ALU

- Earlier in the semester
  - Number representations, 2’s complement, unsigned
  - Addition/Subtraction
  - Add/Sub ALU
    - Full adder, ripple carry, subtraction
  - Carry-lookahead addition
  - Logical operations
    - and, or, xor, nor, shifts
  - Overflow

Basic Arithmetic and the ALU

- Now
  - Integer multiplication
    - Booth's algorithm
- This is not crucial for the project

Multiplication

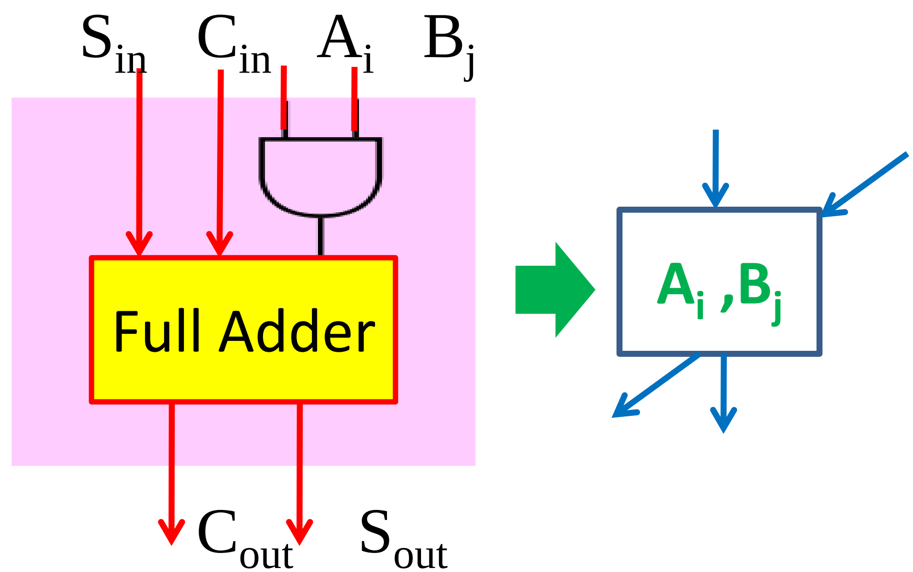
- Flashback to 3rd grade
  - Multiplier
  - Multiplicand
  - Partial products
  - Final sum
- Base 10: 8 x 9 = 72
  - PP: 8 + 0 + 0 + 64 = 72
- How wide is the result?
  - log(n x m) = log(n) + log(m)
  - 32b x 32b = 64b result

```
          1  0  0  0
       x  1  0  0  1
       -------------
          1  0  0  0
       0  0  0  0
    0  0  0  0
 1  0  0  0
--------------------
 1  0  0  1  0  0  0
```

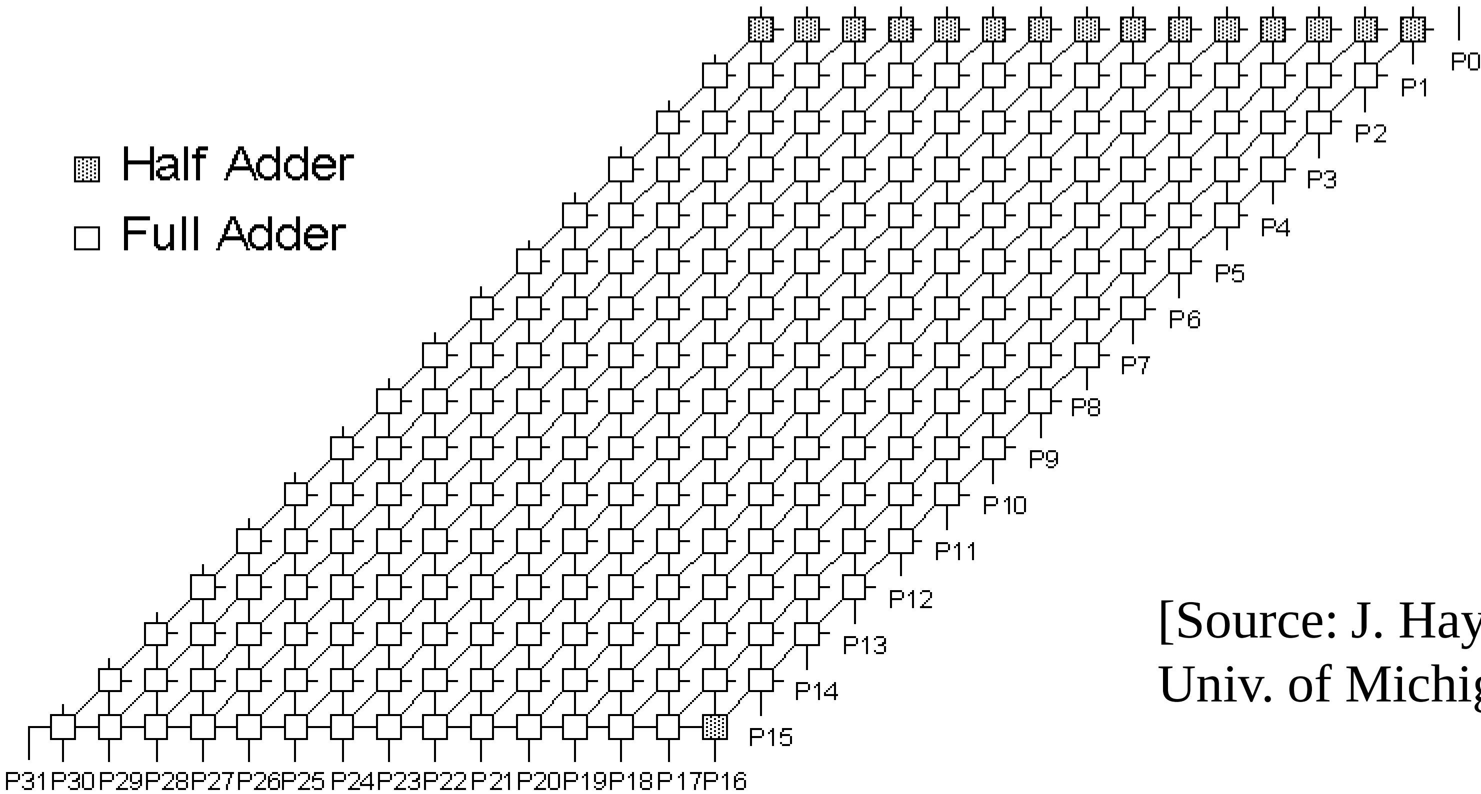
Array Multiplier

			1	0	0	0
		x	1	0	0	1
			1	0	0	0
		0	0	0	0	
	0	0	0	0		
1	0	0	0			
1	0	0	1	0	0	0

- Adding all partial products simultaneously using an array of basic cells



16-bit Array Multiplier



[Source: J. Hayes, Univ. of Michigan]

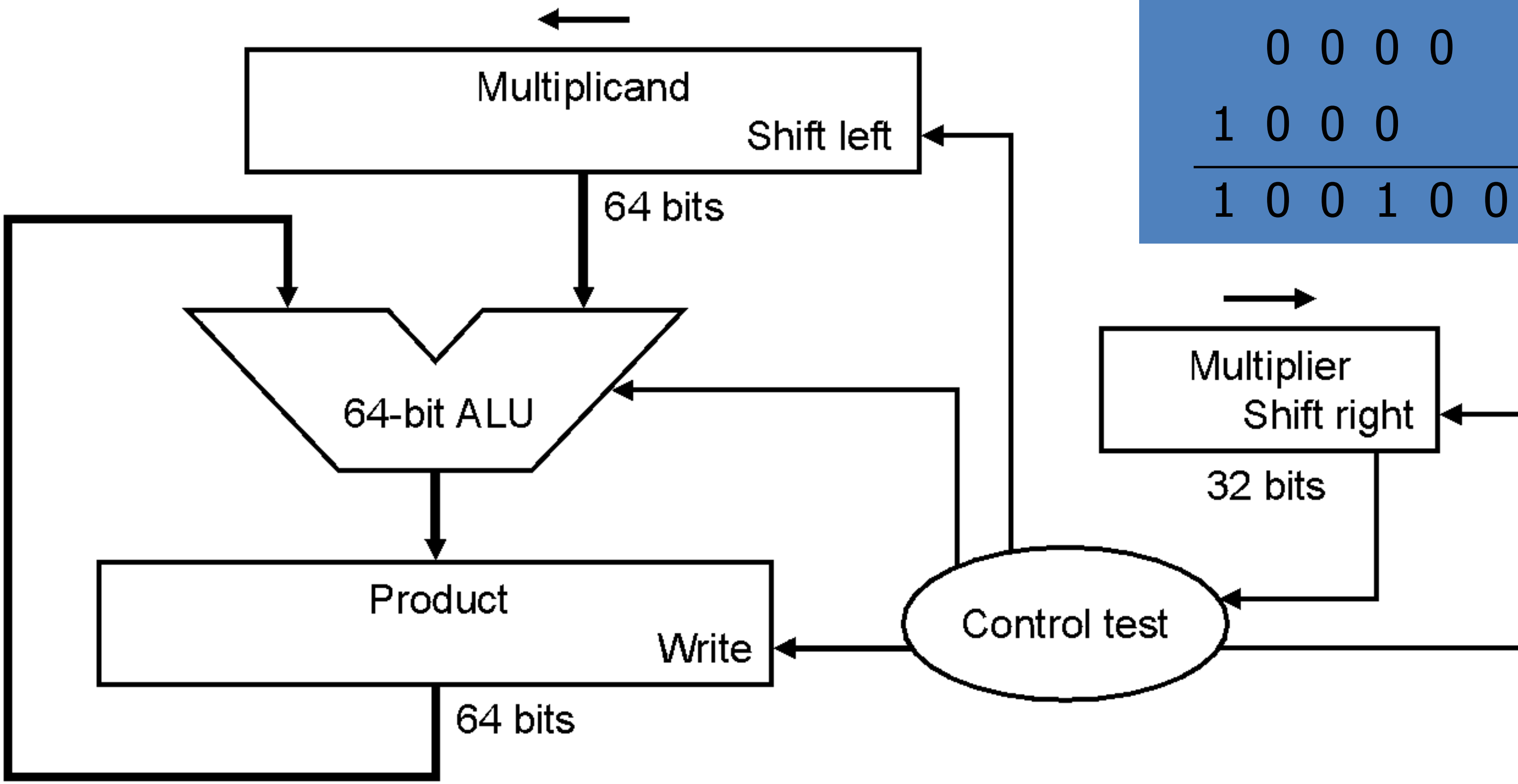
- Conceptually straightforward
- Fairly expensive hardware, integer multiplies relatively rare
  - Most used in array address calc: replace with shifts

Instead: Multicycle Multipliers

- Combinational multipliers
  - Very hardware-intensive
  - Integer multiply relatively rare
  - Not the right place to spend resources
- Multicycle multipliers
  - Iterate through bits of multiplier
  - Conditionally add shifted multiplicand

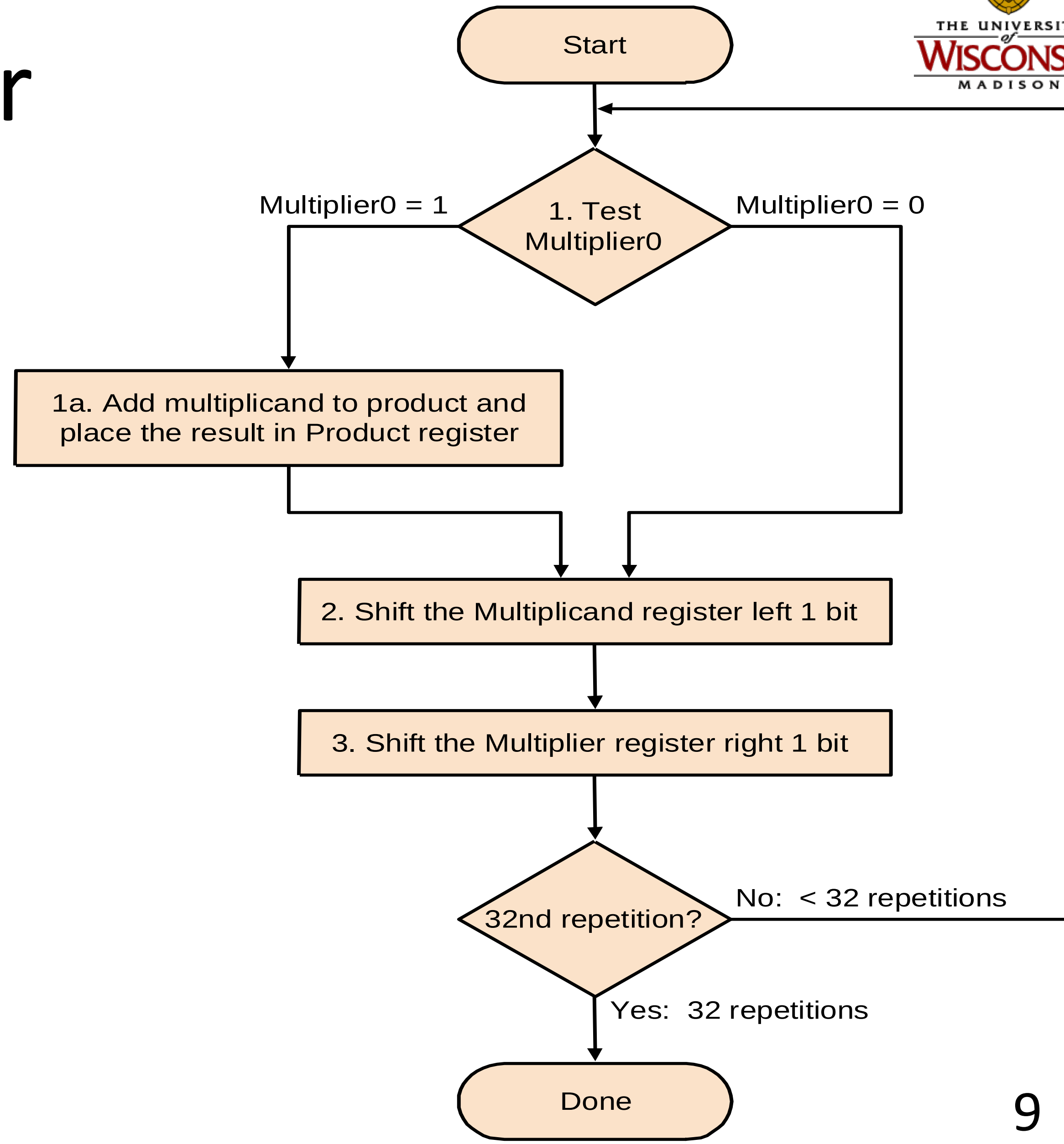
Multiplier

```
      1 0 0 0
    x 1 0 0 1
    ---------
      1 0 0 0
    0 0 0 0
  0 0 0 0
1 0 0 0
-------------
1 0 0 1 0 0 0
```



Multiplier

```
      1 0 0 0
    x 1 0 0 1
    ---------
      1 0 0 0
    0 0 0 0
  0 0 0 0
1 0 0 0
-------------
1 0 0 1 0 0 0
```

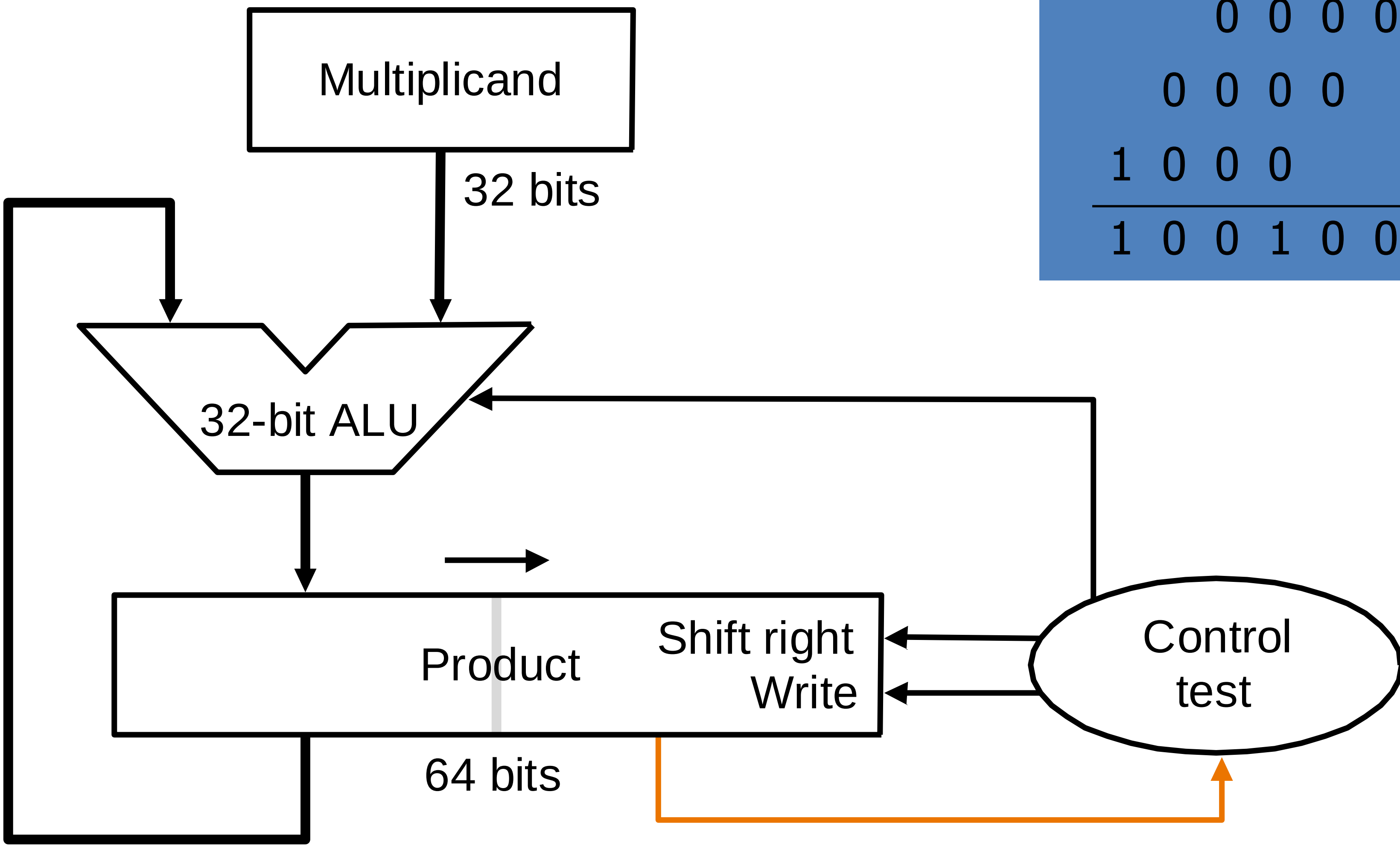


Multiplier Improvements

- Do we really need a 64-bit adder?
  - No, since low-order bits are not involved
  - Hence, just use a 32-bit adder
    - Shift product register right on every step
- Do we really need a separate multiplier register?
  - No, since low-order bits of 64-bit product are initially unused
  - Hence, just store multiplier there initially

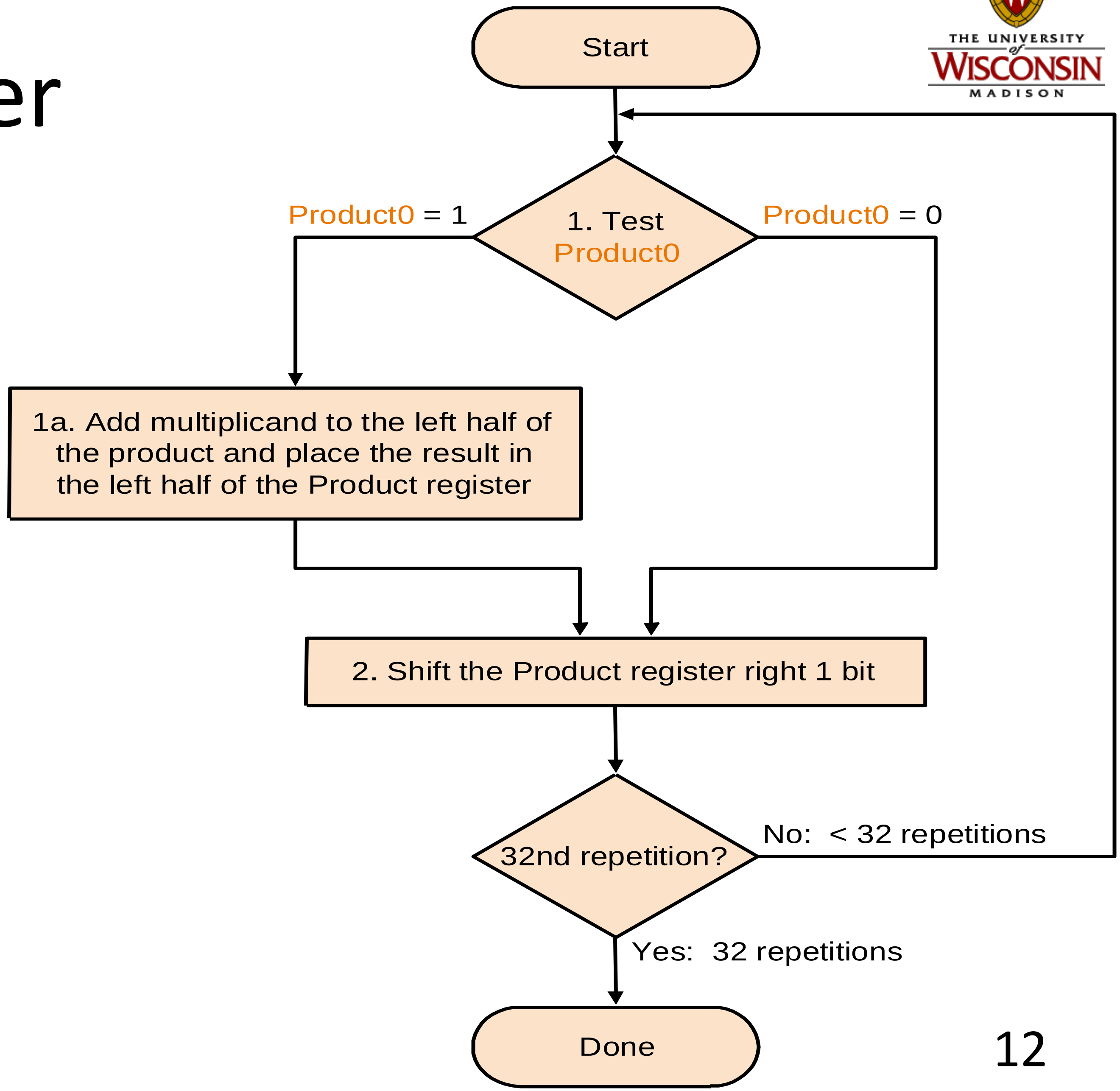
Multiplier

```
       1 0 0 0
     x 1 0 0 1
     ---------
       1 0 0 0
     0 0 0 0
   0 0 0 0
 1 0 0 0
 -------------
 1 0 0 1 0 0 0
```



Multiplier

```
      1 0 0 0
    x 1 0 0 1
    ---------
      1 0 0 0
    0 0 0 0
  0 0 0 0
1 0 0 0
-------------
1 0 0 1 0 0 0
```



Signed Multiplication

- Recall
  - For p = a x b, if a<0 or b<0, then p < 0
  - If a<0 and b<0, then p > 0
  - Hence sign(p) = sign(a) xor sign(b)
- Hence
  - Convert multiplier, multiplicand to positive number with (n-1) bits
  - Multiply positive numbers
  - Compute sign, convert product accordingly
- Or,
  - Perform sign-extension on shifts for prev. design
  - Right answer falls out

Booth's Encoding

- Recall grade school trick
  - When multiplying by 9:
    - Multiply by 10 (easy, just shift digits left)
    - Subtract once
  - E.g.
    - 123454 x 9 = 123454 x (10 – 1) = 1234540 – 123454
    - Converts addition of six partial products to one shift and one subtraction
- Booth's algorithm applies same principle
  - Except no '9' in binary, just '1' and '0'
  - So, it's actually easier!

Booth's Encoding

- Search for a run of '1' bits in the multiplier
  - E.g. '0110' has a run of 2 '1' bits in the middle
  - Multiplying by '0110' (6 in decimal) is equivalent to multiplying by 8 and subtracting twice, since 6 x m = (8 – 2) x m = 8m – 2m
- Hence, iterate right to left and:
  - Subtract multiplicand from product at first '1'
  - Add multiplicand to product after last '1'
  - Don't do either for '1' bits in the middle

Booth’s Algorithm

Current bit	Bit to right	Explanation	Example	Operation
1	0	Begins run of ‘1’	0000111000	Subtract
1	1	Middle of run of ‘1’	0000111000	Nothing
0	1	End of a run of ‘1’	0000111000	Add
0	0	Middle of a run of ‘0’	0000111000	Nothing

Booth's Encoding

- Really just a new way to encode numbers
  - Normally positionally weighted as 2^n
  - With Booth, each position has a sign bit
  - Can be extended to multiple bits

0	1	1	0	Binary
+1	0	-1	0	1-bit Booth
+2		-2		2-bit Booth

2-bits/cycle Booth Multiplier

- For every pair of multiplier bits
  - If Booth’s encoding is ‘-2’
    - Shift multiplicand left by 1, then subtract
  - If Booth’s encoding is ‘-1’
    - Subtract
  - If Booth’s encoding is ‘0’
    - Do nothing
  - If Booth’s encoding is ‘1’
    - Add
  - If Booth’s encoding is ‘2’
    - Shift multiplicand left by 1, then add

2 bits/cycle Booth's

1 bit Booth	
00	+0
01	+M;
10	-M;
11	+0

Current	Previous	Operation	Explanation
00	0	+0;shift 2	[00] => +0, [00] => +0; 2x(+0)+(+0)=+0
00	1	+M; shift 2	[00] => +0, [01] => +M; 2x(+0)+(+M)=+M
01	0	+M; shift 2	[01] => +M, [10] => -M; 2x(+M)+(-M)=+M
01	1	+2M; shift 2	[01] => +M, [11] => +0; 2x(+M)+(+0)=+2M
10	0	-2M; shift 2	[10] => -M, [00] => +0; 2x(-M)+(+0)=-2M
10	1	-M; shift 2	[10] => -M, [01] => +M; 2x(-M)+(+M)=-M
11	0	-M; shift 2	[11] => +0, [10] => -M; 2x(+0)+(-M)=-M
11	1	+0; shift 2	[11] => +0, [11] => +0; 2x(+0)+(+0)=+0

Booth’s Example

- Negative multiplicand:

  -6 x 6 = -36

  1010 x 0110, 0110 in Booth’s encoding is +0-0

  Hence:

1111 1010	x 0	0000 0000
1111 0100	x –1	0000 1100
1110 1000	x 0	0000 0000
1101 0000	x +1	1101 0000
	Final Sum:	1101 1100 (-36)

Booth's Example

- Negative multiplier:

  -6 x -2 = 12

  1010 x 1110, 1110 in Booth's encoding is 00-0

  Hence:

1111 1010	x 0	0000 0000
1111 0100	x –1	0000 1100
1110 1000	x 0	0000 0000
1101 0000	x 0	0000 0000
	Final Sum:	0000 1100 (12)

Summary

- Integer multiply
  - Combinational
  - Multicycle
  - Booth's algorithm

ECE/CS 552: Integer Dividers

© Prof. Mikko Lipasti

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

Basic Arithmetic and the ALU

- Integer division
  - Restoring, non-restoring
- These are not crucial for the project

Integer Division

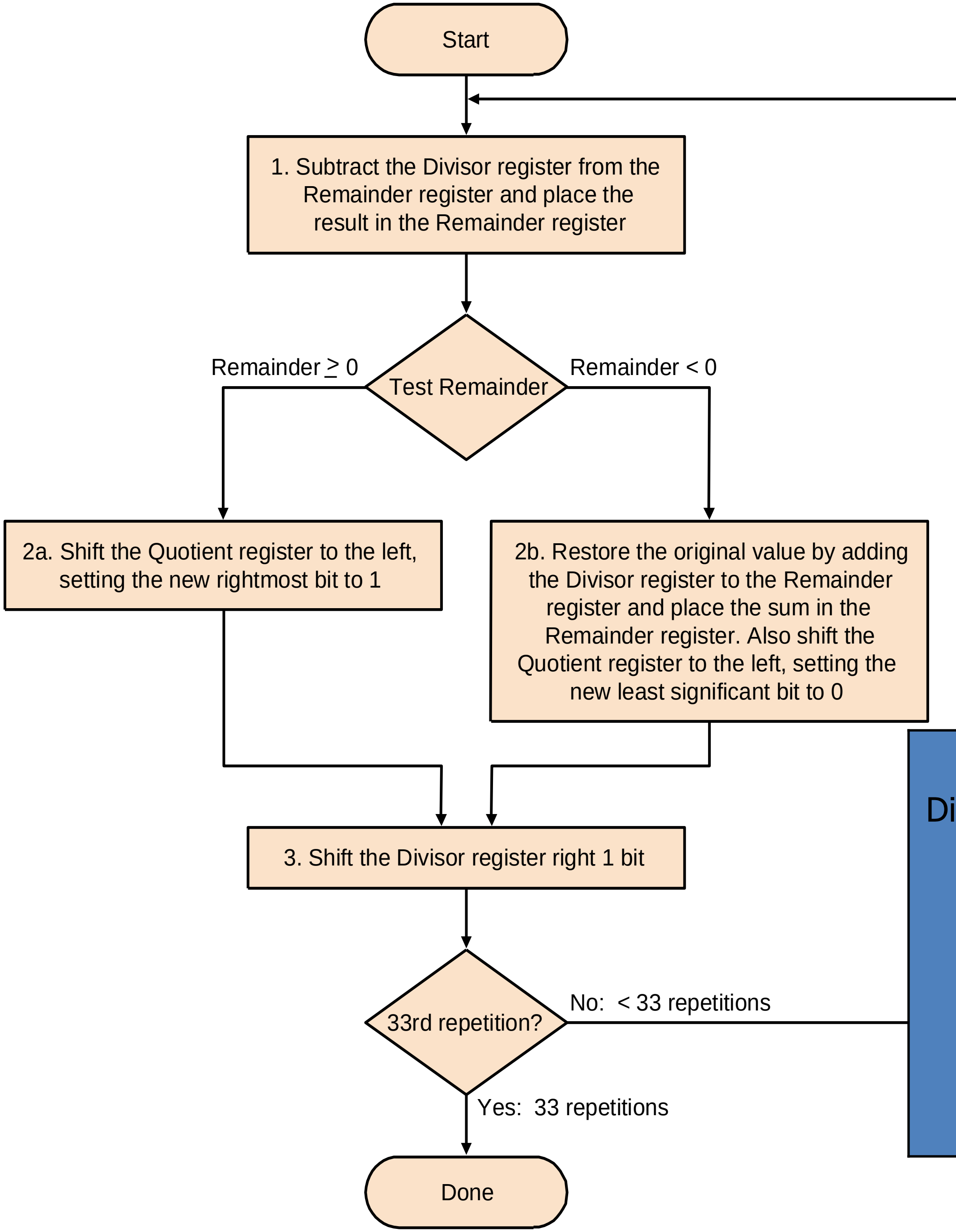
- Again, back to 3rd grade (74 ÷ 8 = 9 rem 2)

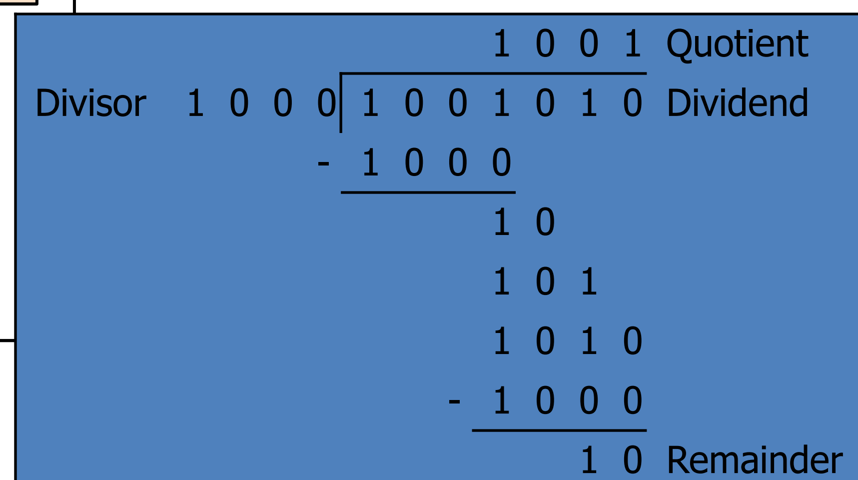
								1	0	0	1	Quotient
Divisor	1	0	0	0	1	0	0	1	0	1	0	Dividend
				-	1	0	0	0				
								1	0			
								1	0	1		
								1	0	1	0	
							-	1	0	0	0	
										1	0	Remainder

Integer Division

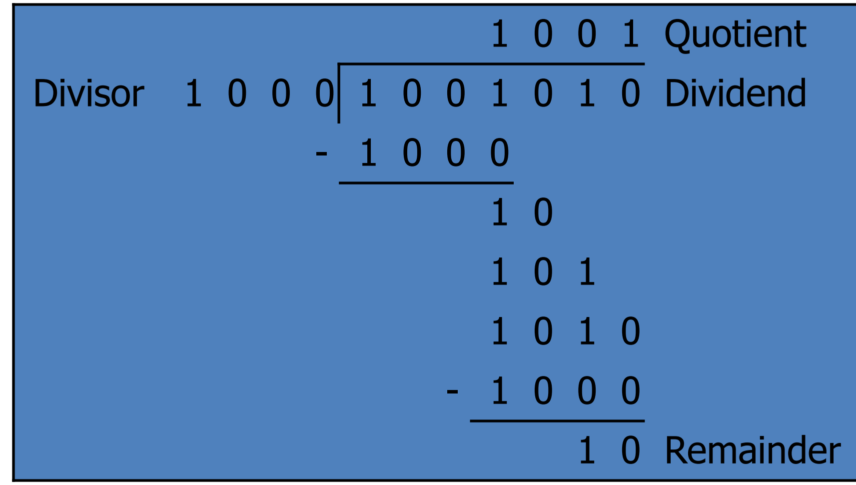
- How does hardware know if division fits?
  - Condition: if remainder ≥ divisor
  - Use subtraction: (remainder – divisor) ≥ 0
- OK, so if it fits, what do we do?
  - $\text{Remainder}_{n+1} = \text{Remainder}_n - \text{divisor}$
- What if it doesn't fit?
  - Have to restore original remainder
- Called restoring division

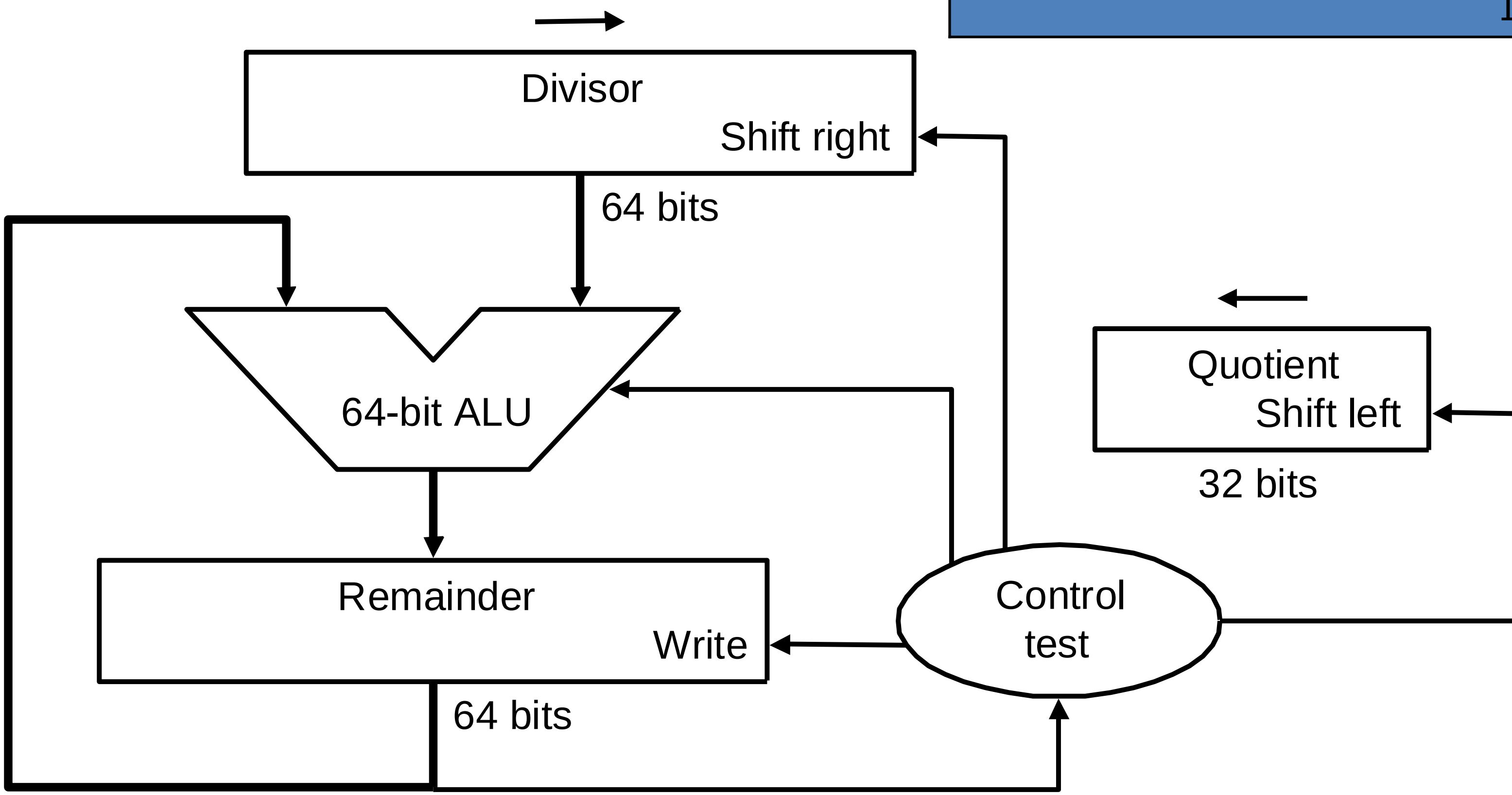
Integer Division





Integer Division

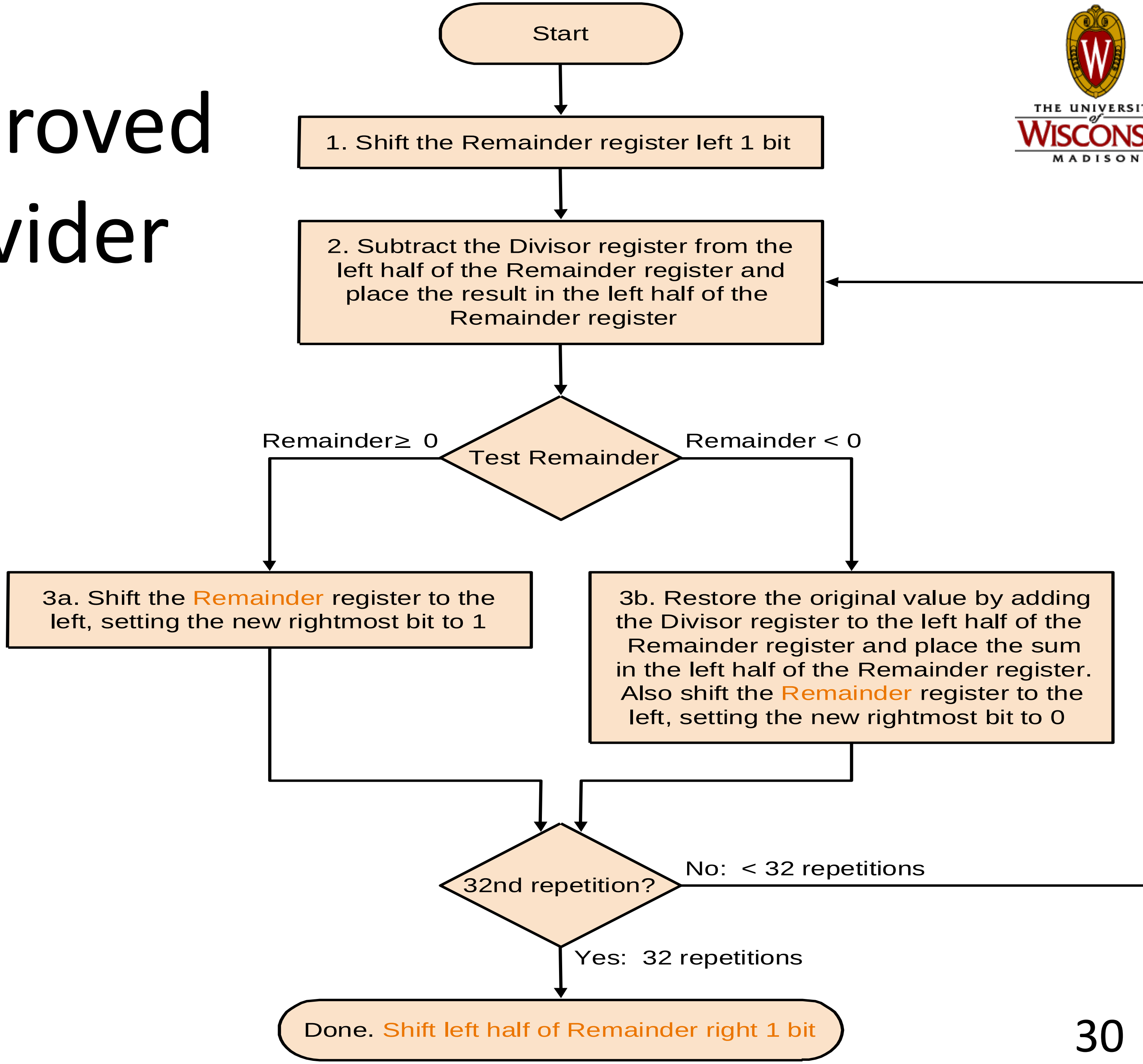




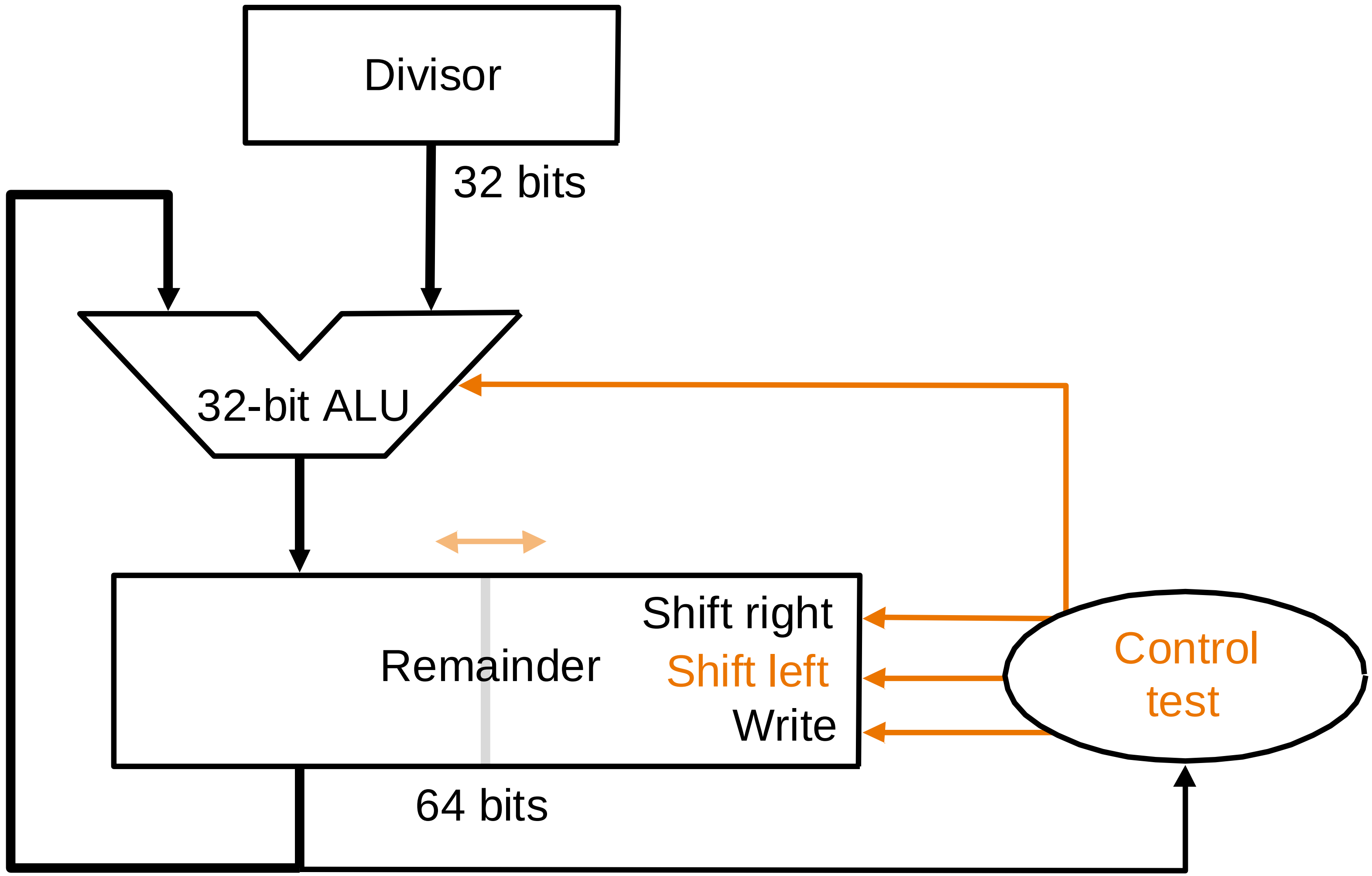
Division Improvements

- Skip first subtract
  – Can’t shift ‘1’ into quotient anyway
  – Hence shift first, then subtract
    - Undo extra shift at end
- Hardware similar to multiplier
  – Can store quotient in remainder register
  – Only need 32b ALU
    - Shift remainder left vs. divisor right

Improved Divider



Improved Divider



Further Improvements

- Division still takes:
  – 2 ALU cycles per bit position
    - 1 to check for divisibility (subtract)
    - One to restore (if needed)
- Can reduce to 1 cycle per bit
  – Called non-restoring division
  – Avoids restore of remainder when test fails

Non-restoring Division

- Consider remainder to be restored:

  $R_i = R_{i-1} - d < 0$

  - Since R_i is negative, we must restore it, right?
  - Well, maybe not. Consider next step i+1:

  $R_{i+1} = 2 \times (R_i) - d = 2 \times (R_i - d) + d$

- Hence, we can compute R_{i+1} by not restoring R_i, and adding d instead of subtracting d
  - Same value for R_{i+1} results
- Throughput of 1 bit per cycle

NR Division Example

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift rem left 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 1110
	3b: Rem < 0 (add next), sll 0	0010	1101 1100
2	2: Rem = Rem + Div	0010	1111 1100
	3b: Rem < 0 (add next), sll 0	0010	1111 1000
3	2: Rem = Rem + Div	0010	0001 1000
	3a: Rem > 0 (sub next), sll 1	0010	0011 0001
4	Rem = Rem – Div	0010	0001 0001
	Rem > 0 (sub next), sll 1	0010	0010 0011
	Shift Rem right by 1	0010	0001 0011

Summary

- Integer dividers covered
  - Multicycle restoring
  - Non-restoring
- Other approaches
  - SRT division [Sweeney, Robertson, Tocher] uses lookup tables
    - Famous Intel fdiv bug caused by incomplete table
  - Newton-Raphson method
    - Estimate reciprocal, iterate to refine, multiply
  - Beyond the scope of this course

ECE/CS 552: Floating Point

© Prof. Mikko Lipasti

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

Basic Arithmetic and the ALU

- Now
  - Floating point representation
  - Floating point addition, multiplication
- These are not crucial for the project

Floating Point

- Want to represent larger range of numbers
  - Fixed point (integer): $-2^{n-1} \ldots (2^{n-1} - 1)$
- How? Sacrifice precision for range by providing exponent to shift relative weight of each bit position
- Similar to scientific notation:

  3.14159×10^{23}
- Cannot specify every discrete value in the range, but can span much larger range

Floating Point

- Still use a fixed number of bits
  - Sign bit S, exponent E, significand F
  - Value: $(-1)^S \times F \times 2^E$
- IEEE 754 standard

	Size	Exponent	Significand	Range
Single precision	32b	8b	23b	$2 \times 10^{+/-38}$
Double precision	64b	11b	52b	$2 \times 10^{+/-308}$

Floating Point Exponent

- Exponent specified in *biased* or *excess* notation
- Why?
  - To simplify sorting
  - Sign bit is MSB to ease sorting
  - 2's complement exponent:
    - Large numbers have positive exponent
    - Small numbers have negative exponent
  - Sorting does not follow naturally

Excess or Biased Exponent

Exponent	2's Compl	Excess-127
-127	1000 0001	0000 0000
-126	1000 0010	0000 0001
…	…	…
+127	0111 1111	1111 1110

- Value: $(-1)^S \times F \times 2^{(E\text{-bias})}$
  - SP: bias is 127
  - DP: bias is 1023

Floating Point Normalization

- S,E,F representation allows more than one representation for a particular value, e.g.

  $1.0 \times 10^5 = 0.1 \times 10^6 = 10.0 \times 10^4$

  - This makes comparison operations difficult
  - Prefer to have a single representation
- Hence, normalize by convention:
  - Only one digit to the left of the floating point
  - In binary, that digit must be a 1
    - Since leading '1' is implicit, no need to store it
    - Hence, obtain one extra bit of precision for free

FP Overflow/Underflow

- FP Overflow
  - Analogous to integer overflow
  - Result is too big to represent
  - Means exponent is too big
- FP Underflow
  - Result is too small to represent
  - Means exponent is too small (too negative)
- Both can raise an exception under IEEE754

IEEE754 Special Cases

Single Precision		Double Precision		Value
Exponent	Significand	Exponent	Significand	
0	0	0	0	0
0	nonzero	0	nonzero	denormalized
1-254	anything	1-2046	anything	fp number
255	0	2047	0	infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

FP Rounding

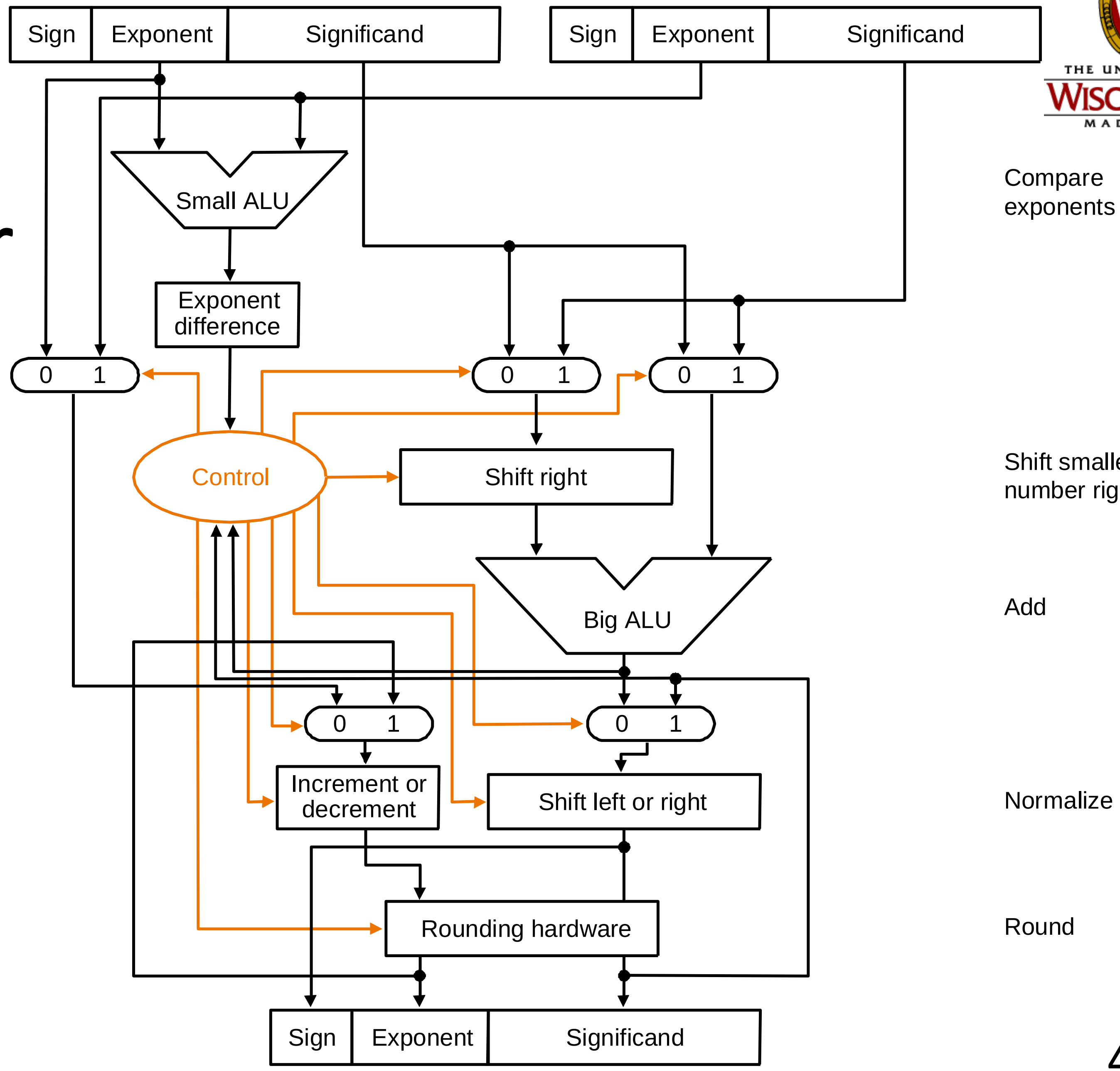
- Rounding is important
  - Small errors accumulate over billions of ops
- FP rounding hardware helps
  - Compute extra guard bit beyond 23/52 bits
  - Further, compute additional round bit beyond that
    - Multiply may result in leading 0 bit, normalize shifts guard bit into product, leaving round bit for rounding
  - Finally, keep sticky bit that is set whenever ‘1’ bits are “lost” to the right
    - Differentiates between 0.5 and 0.500000000001

Floating Point Addition

- Just like grade school
  - First, align decimal points
  - Then, add significands
  - Finally, normalize result
- Example

9.997×10^2	9.997000×10^2
4.631×10^{-1}	0.004631×10^2
Sum	10.001631×10^2
Normalized	1.0001631×10^3

FP Adder



Sign
Exponent
Significand
Sign
Exponent
Significand
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Significand
Compare exponents
Shift smaller number right
Add
Normalize
Round

FP Multiplication

- Sign: $P_s = A_s \text{ xor } B_s$
- Exponent: $P_E = A_E + B_E$
  - Due to bias/excess, must subtract bias

    e = e1 + e2

    E = e + 1023 = e1 + e2 + 1023

    E = (E1 – 1023) + (E2 – 1023) + 1023

    E = E1 + E2 –1023
- Significand: $P_F = A_F \times B_F$
  - Standard integer multiply (23b or 52b + g/r/s bits)
  - Use Wallace tree of CSAs to sum partial products

FP Multiplication

- Compute sign, exponent, significand
- Normalize
  - Shift left, right by 1
- Check for overflow, underflow
- Round
- Normalize again (if necessary)

Summary

- Floating point representation
  - Normalization
  - Overflow, underflow
  - Rounding
- Floating point add
- Floating point multiply