EECS150 - Digital Design

Lecture 22 - Arithmetic and Logic Circuits Part 3

April 14, 2005

John Wawrzynek

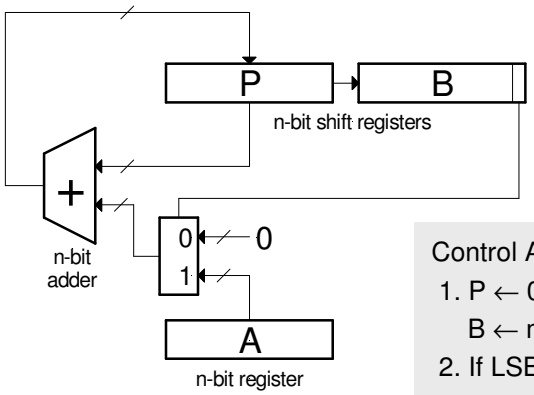
Multiplication

$$
\begin{array}{llllll}
 & & a_3 & a_2 & a_1 & a_0 \leftarrow \textit{Multiplicand} \\
 & & b_3 & b_2 & b_1 & b_0 \leftarrow \textit{Multiplier} \\
\hline
 & X & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & \\
\hline
 & & \ldots & a_1b_0+a_0b_1 & a_0b_0 & \leftarrow \textit{Product}
\end{array}
$$

Partial products

Many different circuits exist for multiplication. Each one has a different balance between speed (performance) and amount of logic (cost).

"Shift and Add" Multiplier



- Sums each partial product, one at a time.
- In binary, each partial product is shifted versions of A or 0.

- Cost α n, T = n clock cycles.
- What is the critical path for determining the min clock period?

Control Algorithm:

1. P ← 0, A ← multiplicand, B ← multiplier
2. If LSB of B==1 then add A to P else add 0
3. Shift [P][B] right 1
4. Repeat steps 2 and 3 n-1 times.
5. [P][B] has product.

"Shift and Add" Multiplier

Signed Multiplication:

Remember for 2's complement numbers MSB has negative weight:

$$X = \sum_{i=0}^{N-2} x_i 2^i - x_{n-1} 2^{n-1}$$

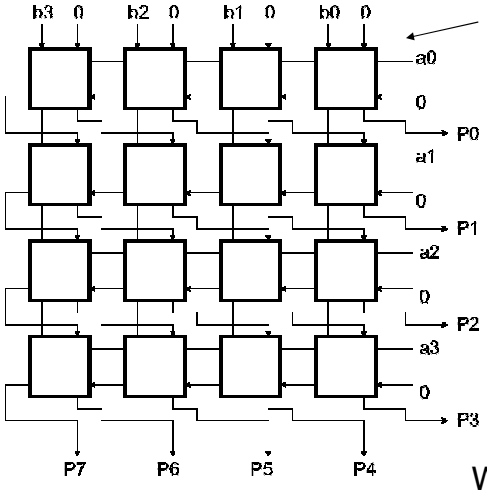
ex: -6 = 11010_2 = $0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 - 1 \cdot 2^4$

= 0 + 2 + 0 + 8 - 16 = -6

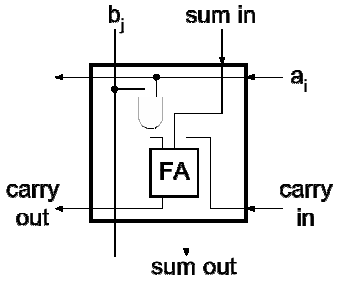
- Therefore for multiplication:
  - a) subtract final partial product
  - b) sign-extend partial products
- Modifications to shift & add circuit:
  - a) adder/subtractor
  - b) sign-extender on P shifter register

Array Multiplier

Generates all n partial products simultaneously.



Each row: n-bit adder with AND gates



What is the critical path?

Carry-save Addition

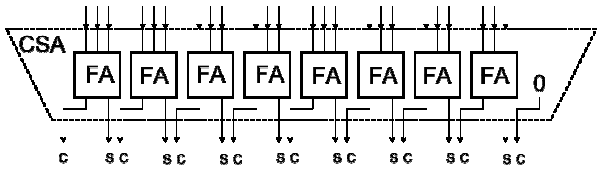
- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- "Carry-save" addition can help.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.

- Example: sum three numbers,
  $3_{10} = 0011$, $2_{10} = 0010$, $3_{10} = 0011$

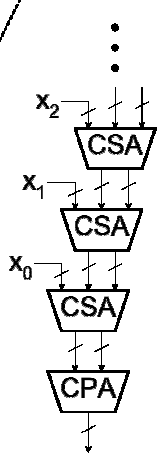
$$
\begin{array}{rll}
 & 3_{10}\ 0011 & \\
+ & 2_{10}\ \underline{0010} & \text{carry-save add} \\
 & c\ 0100 = 4_{10} & \\
 & s\ 0001 = 1_{10} & \\
 & 3_{10}\ \underline{0011} & \text{carry-save add} \\
 & c\ 0010 = 2_{10} & \\
 & s\ \underline{0110} = 6_{10} & \text{carry-propagate add} \\
 & \quad 1000 = 8_{10} &
\end{array}
$$

- In general, *carry-save* addition takes in 3 numbers and produces 2.
- Whereas, *carry-propagate* takes 2 and produces 1.
- With this technique, we can avoid carry propagation until final addition

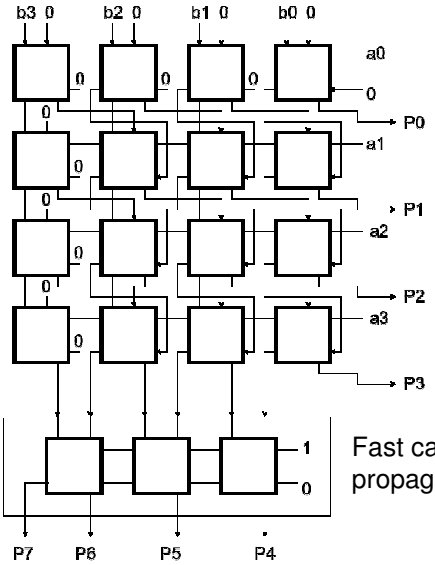
Carry-save Circuits



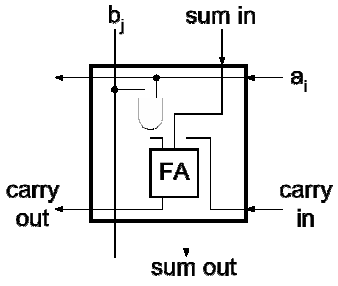
- When adding sets of numbers, carry-save can be used on all but the final sum.
- Standard adder (carry propagate) is used for final sum.



Array Multiplier using Carry-save Addition



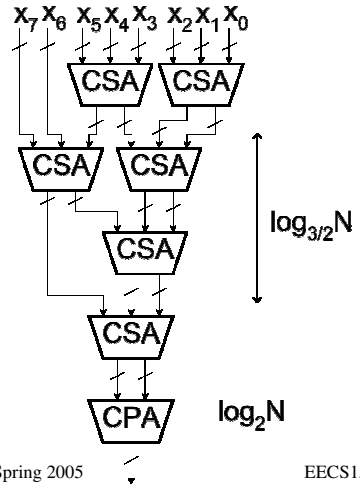
Fast carry-propagate adder



Carry-save Addition

CSA is associative and communitive. For example:

$$(((X_0 + X_1) + X_2) + X_3) = ((X_0 + X_1) + (X_2 + X_3))$$



- A balanced tree can be used to reduce the logic delay.
- This structure is the basis of the ***Wallace Tree Multiplier***.
- Partial products are summed with the CSA tree. Fast CPA (ex: CLA) is used for final sum.
- Multiplier delay α $\log_{3/2}N + \log_2 N$

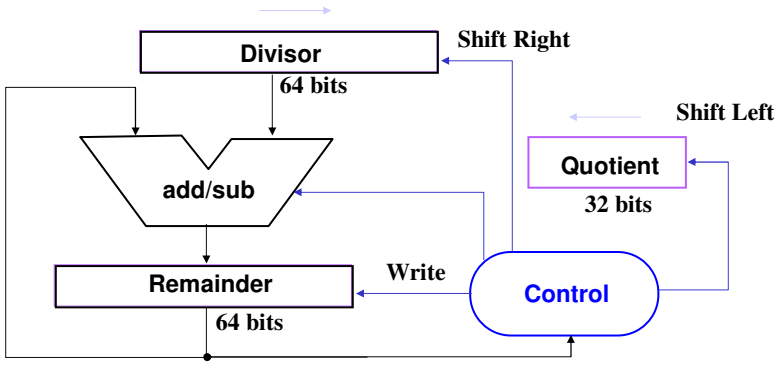
Division

```
                    1001   Quotient
Divisor   1000 | 1001010   Dividend
               -1000
                   10
                  101
                 1010
                -1000
                   10   Remainder (or Modulo result)
```

- See how big a number can be subtracted, creating quotient bit on each step
  - Binary $\Rightarrow$ 1 * divisor or 0 * divisor
- Dividend = Quotient x Divisor + Remainder
  - sizeof(dividend) = sizeof(quotient) + sizeof(divisor)
- 3 versions of divide, "successive refinement"

DIVIDE HARDWARE Version 1

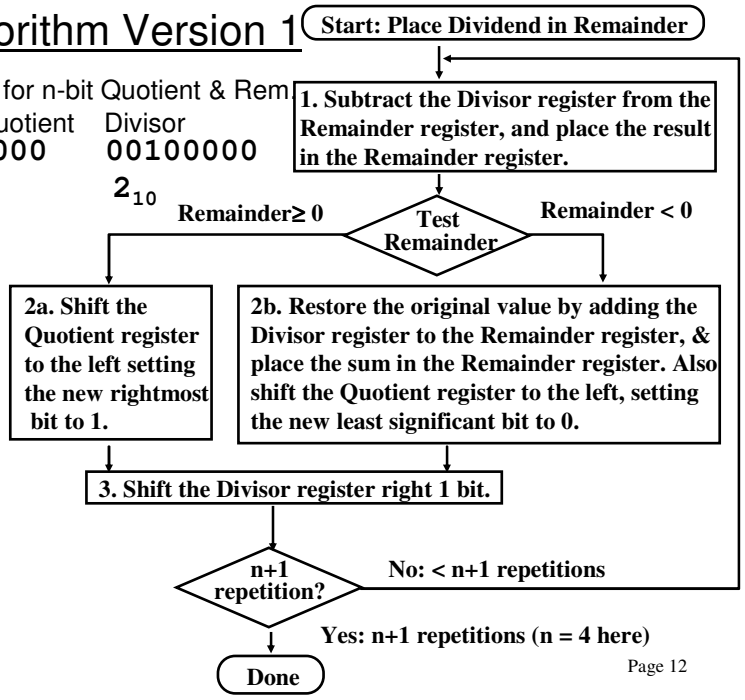
- 64-bit Divisor register, 64-bit adder/subtractor, 64-bit Remainder register, 32-bit Quotient register



Divide Algorithm Version 1

Takes n+1 steps for n-bit Quotient & Rem

Remainder	Quotient	Divisor
00000111	0000	00100000
7_{10}		2_{10}



Version 1 Division Example 7/2

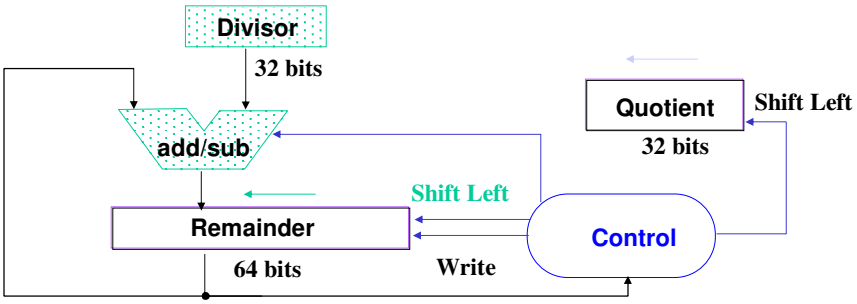
Iteration step		quotient	divisor	remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: rem=rem-div	0000	0010 0000	1110 0111
	2b: rem<0 ⇒ +div, sll Q, Q0=0	0000	0010 0000	0000 0111
	3: shift div right	0000	0001 0000	0000 0111
2	1: rem=rem-div	0000	0001 0000	1111 0111
	2b: rem<0 ⇒ +div, sll Q, Q0=0	0000	0001 0000	0000 0111
	3: shift div right	0000	0000 1000	0000 0111
3	1: rem=rem-div	0000	0000 1000	1111 1111
	2b: rem<0 ⇒ +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≥0 ⇒ 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≥0 ⇒ sll Q, Q0=1	0011	0000 0010	0000 0001
	3: shift div right	0011	0000 0001	0000 0001

Observations on Divide Version 1

- 1/2 bits in divisor always 0
  ⇒ 1/2 of 64-bit adder is wasted
  ⇒ 1/2 of divisor is wasted
- Instead of shifting divisor to right,
  shift remainder to left?
- 1st step cannot produce a 1 in quotient bit
  (otherwise quotient ≥ 2^n)
  ⇒ switch order to shift first and then subtract,
  can save 1 iteration

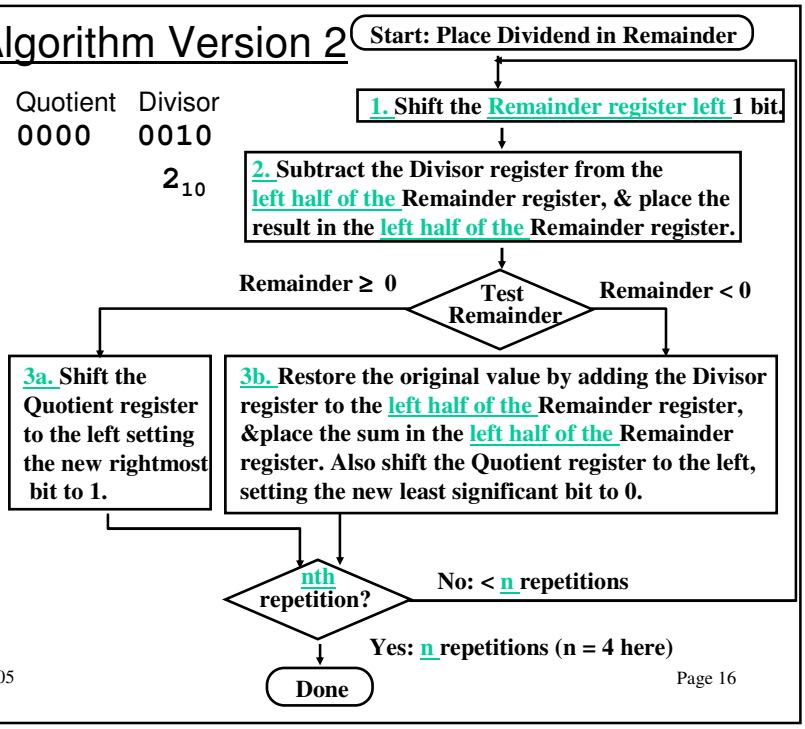
DIVIDE HARDWARE Version 2

- 32-bit Divisor register, 32-bit ALU, 64-bit Remainder register, 32-bit Quotient register



Divide Algorithm Version 2

Remainder	Quotient	Divisor
00000111	0000	0010
7_{10}		2_{10}

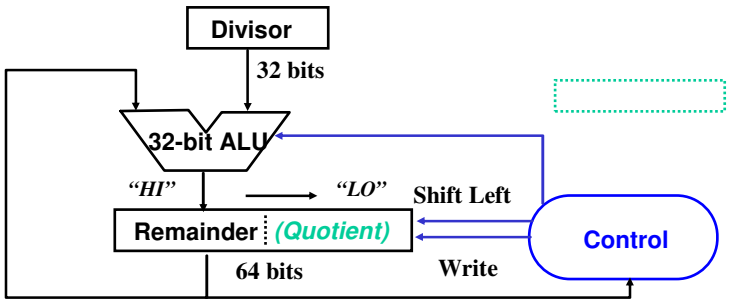


Observations on Divide Version 2

- Eliminate Quotient register by combining with Remainder as shifted left.
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register

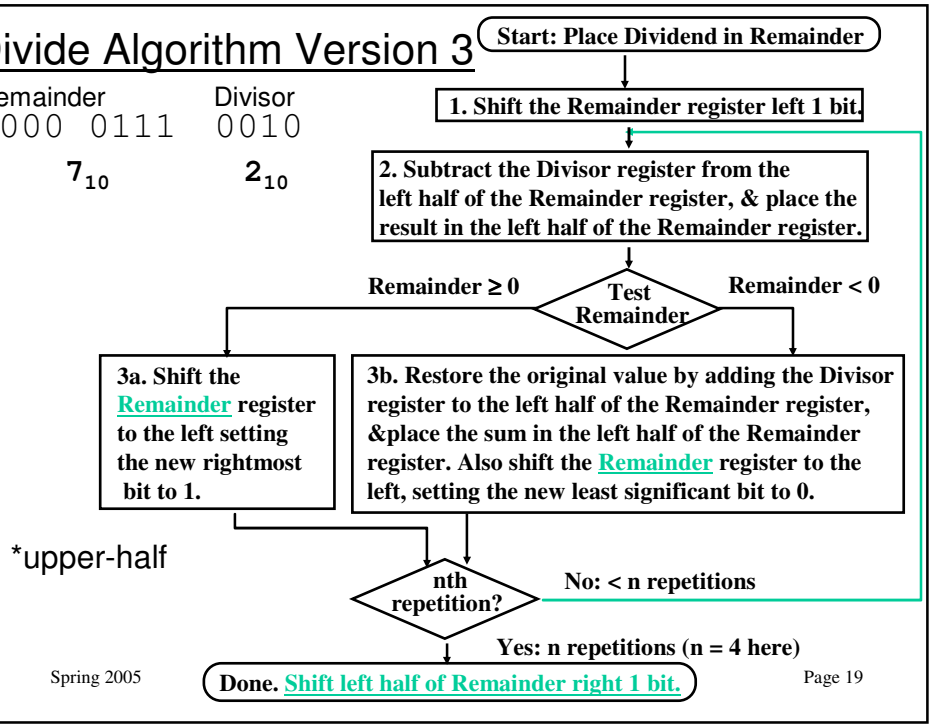
DIVIDE HARDWARE Version 3

- 32-bit Divisor register, 32-bit adder/subtractor, 64-bit Remainder register, (0-bit Quotient reg)



Divide Algorithm Version 3

Remainder 0000 0111 7_{10} Divisor 0010 2_{10}



Observations on Divide Version 3

- Same Hardware as shift and add multiplier: just 63-bit register to shift left or shift right
- Signed divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree e.g., –7 ÷ 2 = –3, remainder = –1
- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits (“called saturation”)