CPE 626 CPU Resources: Multipliers

Aleksandar Milenkovic
E-mail: milenka@ece.uah.edu
Web: http://www.ece.uah.edu/~milenka

Outline

- **Unsigned Multiplication**
- **Shift and And Multiplier / Divider**
- **Speeding Up Multiplication**
- **Array Multiplier**
- **Signed Multiplication**
- **Booth Encoding**
- **Wallace-tree**

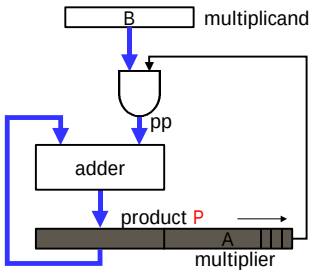
Unsigned Multiplication

```
      011101   multiplicand (29)
  x   101011   multiplier (43)
  ------------------------
      011101 ←  partial product
     011101  ←
    000000   ←
   011101    ←
  000000     ←
 011101      ←
  ------------------------------
 1001110111111 ← product
```

- product = 0
- for i = 0 to n–1
  - compute partial product (AND operation)
  - left-shift partial product by *i*
  - product += partial product

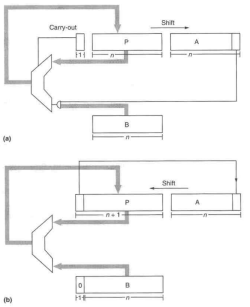
Shift and Add Multiplier

- for i = 0 to n–1
  - pp = B · a[0]
  - P[2n–1:n] += pp
  - P = P >> 1



Shift and Add Multiplier/Divider

- (a) Multiplier (b) Divider
- Operands:
  n-bit unsigned integers
- Multiply steps (n steps)
  - if (A(0) == 1) P <= P + B
    else P <= P + 0
  - P and A are shifted right with carry out of the sum being moved into the MSB of P, the LSB of P moved into MSB of A and LSB of A being shifted out



Division

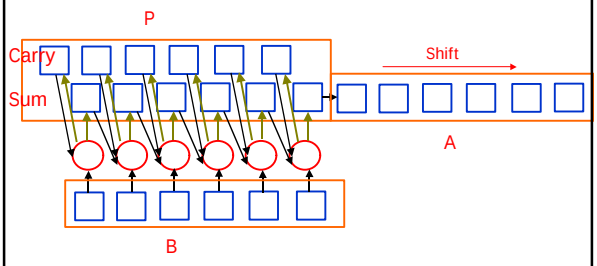
- Operands (a/b):
  n-bit unsigned integers
  - put a in register A
  - put b in register B
  - put 0 in register P
- Divide steps (n steps)
  - Shift (P, A) register pair one bit left
  - P <= P – B
  - if result is negative, set the low order bit of A to 0, otherwise to 1
  - if the result of step 2 is negative, restore the old value of P by adding the contents of B back to P

P	A	
00000	1110	Divide $14 = 1110_2$ by $3 = 11_2$. B always contains 0011_2.
00001	110	step 1(i): shift.
–00011		step 1(ii): subtract.
–00010	1100	step 1(iii): result is negative, set quotient bit to 0.
00001	1100	step 1(iv): restore.
00011	100	step 2(i): shift.
–00011		step 2(ii): subtract.
00000	1001	step 2(iii): result is nonnegative, set quotient bit to 1.
00001	001	step 3(i): shift.
–00011		step 3(ii): subtract.
–00010	0010	step 3(iii): result is negative, set quotient bit to 0.
00001	0010	step 3(iv): restore.
00010	010	step 4(i): shift.
–00011		step 4(ii): subtract.
–00001	0100	step 4(iii): result is negative, set quotient bit to 0.
00010	0100	step 4(iv): restore. The quotient is 0100_2 and the remainder is 00010_2.

Speeding Up Multiplication (cont'd)

- Reduce the amount of computation in each step by using carry-save adders (CSA)
- CSA is simply collection of n independent full adders
- Each addition operation results in a pair of bits, stored in the sum and carry parts of P
- At each step, only the LSB bit of the sum needs to be shifted
- Steps
  - load the sum and carry bits of P with zero
  - perform first addition
  - shift the LSB sum bit of P into A, as well as A itself
    Note: (n-1) bit of P do not need to be shifted because on the next cycle the sum bits are fed into the next lower order adder
- Disadvantages
  - Additional hardware (keep both carry and sum)
  - After the last step, the high order word of the result must be fed into an ordinary adder to combine the sum and carry parts

Speeding Up Multiplication



An Example

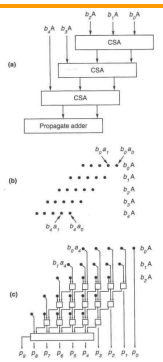
- 9 x 5 => 1001 x 0101 = 0010 1101
  - C = 0000
    S = 0000 A = 0101
    P = 1001
  - C = 0000
    S = 1001 A = 1010
    P = 0000
  - C = 0000
    S = 0100 A = 0101
    P = 1001
  - C = 0000
    S = 1011 A = 1010
    P = 0000
  - Carry Propagate
    C = 0000
    S = 0101 A = 1101
    S = 0010 A = 1101

Speeding Up Multiplication (cont'd)

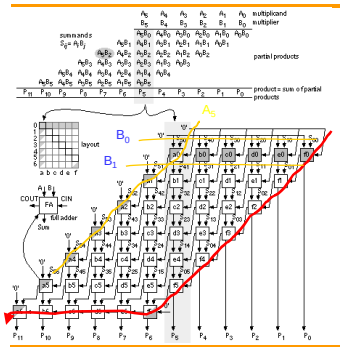
- Another approach is to examine k low order bits of A at each step, rather than just one bit
  => *higher-radix multiplication*
- Radix-4 Booth recoding
- Radix-8 Booth recoding
- ...

Array Multiplier

- If the space for many adders is available, then multiplication speed can be improved
- E. g. 5-bit multiplier (3 CSA + CPA)
- Advantage
  - could be pipelined
- If space budget is limited, use multiple-pass arrangements

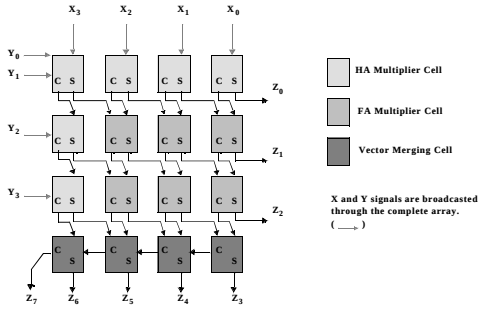


6-bit Array Multiplier

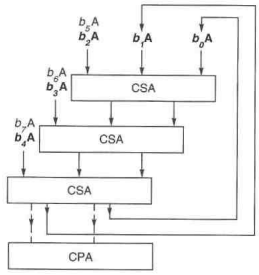


- Adders a0-f0 may be eliminated => this eliminates adders a1-a6
- Complexity:
  CSA - 5x6 adders (including 5 half adders)
  CPA – 6 adders (2 HAs)
- Delay:
  proportional to n + delay of CPA (f6 – b6)
- How to improve performance?
  - decrease the number of partial products
  - improve the speed of the addition of the partial

Floorplan of the 4-bit Array Multiplier

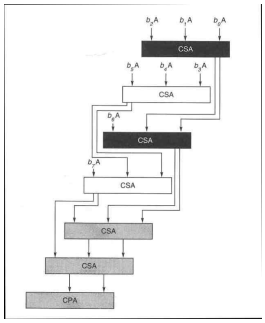


Multipass Array Multiplier



Even/odd Array

- First two adders work in parallel
- Their results are fed into third and fourth adders, which also work in parallel



Using CSD Vector

- 15 (multiplicand) x 19 (multiplier) = ?

$$15 \times 19 = 15 \times (20 - 1) = 15 \times 2\bar{1}$$

- A x B, B = 00010111
  - B = 16 + 4 + 2 + 1 = 23
  - Computation: 4 add operations
- It is easier to multiply A with the canonical signed-digit vector (CSD vector) D

$$D = 0010\bar{1}00\bar{1} = 32 - 8 - 1 = 23$$

  - Computation: 3 add/sub operations (a subtraction is as easy as an addition)
- Weight – number of partial products by 1: B has 4, D has 3

CSD Vector

- Recode (or encode) any binary number, B, as a CSD vector D

$D_i = B_i + C_i - 2C_{i+1}$

$C_{i+1} = \text{Carry}\{B_{i+1} + B_i + C_i\},\ C_0 = 0$

$B = 011$

$B_2 = 0, B_1 = 1, B_0 = 1$

$C_1 = \text{Carry}\{1 + 1 + 0\} = 1,\ D_0 = 1 + 0 - 2 = \bar{1}$

$C_2 = \text{Carry}\{0 + 1 + 1\} = 1,\ D_1 = 1 + 1 - 2 = 0$

$C_3 = \text{Carry}\{0 + 0 + 1\} = 0,\ D_2 = 0 + 1 - 0 = 1$

CSD Vector

- N – (n + 1)-digit 2's complement number
- Recode it using a Radix other than 2

$B = B_0 \cdot 2^0 + B_1 \cdot 2^1 + B_2 \cdot 2^2 + \ldots + B_{n-1} \cdot 2^{n-1} - B_n \cdot 2^n$

$B = 2 \cdot B - B$

$= -B_0 \cdot 2^0 + (B_0 - B_1) \cdot 2^1 + \ldots + (B_{i-1} - B_i) \cdot 2^i + \ldots + (B_{n-1} - B_n) \cdot 2^n$

$= (-2 \cdot B_1 + B_0) \cdot 2^0 + (-2 \cdot B_3 + B_2 + B_1) \cdot 2^2 + (-2 \cdot B_5 + B_4 + B_3) \cdot 2^4$

$+ \ldots + (-2 \cdot B_i + B_{i-1} + B_{i-2}) \cdot 2^{i-1} + (-2 \cdot B_{i+2} + B_{i+1} + B_i) \cdot 2^{i+1} + \ldots$

$+ (-2 \cdot B_n + B_{n-1} + B_{n-2}) \cdot 2^{n-1}$

CSD Vector: An Example – Radix = 2

- B = 101001, n = 5

$B = 1 + 8 - 32 = -23$

$B = (0 - B_0) \cdot 2^0 + (B_0 - B_1) \cdot 2^1 + (B_1 - B_2) \cdot 2^2 + (B_2 - B_3) \cdot 2^3$

$+ (B_3 - B_4) \cdot 2^4 + (B_4 - B_5) \cdot 2^5$

$= (-1) + 2 + 0 + (-8) + 16 + (-32) = -23$

$E = \bar{1}1\bar{1}01\bar{1} = (-1) \cdot 2^5 + 1 \cdot 2^4 + (-1) \cdot 2^3 + 1 \cdot 2^1 + (-1) \cdot 2^0$

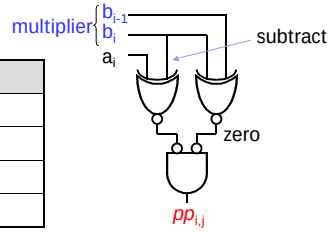
$= -32 + 16 - 8 + 2 - 1 = -23$

- To multiply by B
  - encode it as a radix-2 signed digit E
  - Multiply by 2 (a shift) + 6 (n+1) add/subtract operations

Encoded Partial Products

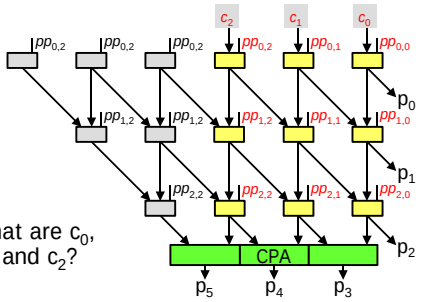
$B = (0 - B_0) \cdot 2^0 + (B_0 - B_1) \cdot 2^1 + \ldots + (B_{i-1} - B_i) \cdot 2^i + \ldots + (B_{n-1} - B_n) \cdot 2^n$

$b_i\ b_{i-1}$	operation
00	do nothing
01	add A
10	subtract A
11	do nothing



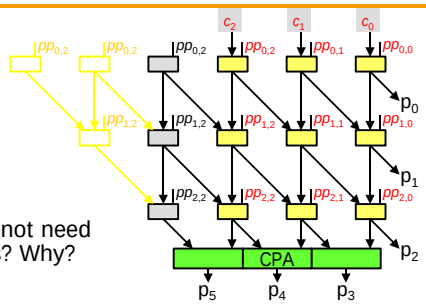
(partial product row i, bit j)

Signed Multiplication (1)



- What are c_0, c_1, and c_2?

Signed Multiplication (2)



- Do not need this? Why?

CSD Vector: An Example – Radix=4

- B = 101001, n = 5

$B = 1 + 8 - 32 = -23$

$B = (-2 \cdot B_1 + B_0) \cdot 2^0 + (-2 \cdot B_3 + B_2 + B_1) \cdot 2^2 + (-2 \cdot B_5 + B_4 + B_3) \cdot 2^4$

$= (2 \cdot 0 + 1) \cdot 2^0 + (-2 \cdot 1 + 0 + 0) \cdot 2^2 + (-2 \cdot 1 + 0 + 1) \cdot 2^4$

$= 1 - 8 - 16 = -23$

$E = 12\bar{1}\bar{1} = 1 \cdot 4^0 + (-2) \cdot 4^1 + (-1) \cdot 4^2$

$= 1 - 8 - 16 = -23$

- To multiply by B
  - encode it as a radix-4 signed digit E
  - Multiply by 4 (a shift by 2) + 3 add/subtract operation

Booth Encoding (1)

- Encode a number by taking groups of 3 bits where each 3-bit group overlaps by 1 bit

$E_j = -2 \cdot B_i + B_{i-1} + B_{i-2}$

$E_{j+1} = -2 \cdot B_{i+2} + B_{i+1} + B_i$

- Consider multiplier B with (n + 1) bit
  - Pad B with 0 to match the first term
  - if B has an odd number of bits, then extend the sign $B_n B_n B_{n-1} \ldots B_0 0$

$B = 01011_2 \Rightarrow B = 8 + 2 + 1 = 11_{10}$

$B = 010110 \Rightarrow B = 0010110 \Rightarrow 001,101,110$

$E = 1\bar{1}\bar{1} = 1 \cdot 4^2 + (-1) \cdot 4^1 + (-1) \cdot 4^0 = 16 - 4 - 1 = 11$

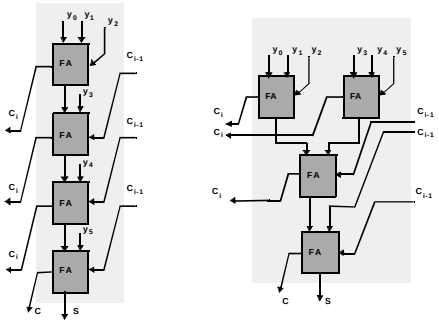
Booth Encoding (2)

B_i	B_{i-1}	B_{i-2}	Operation
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Booth Multiply: An Example

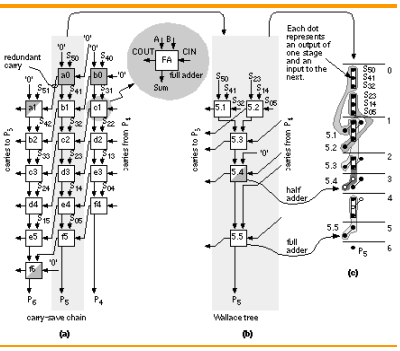
- A = 1100, B = 0111, 2's compl., n = 3
- M = A*B = ?
- B=0111.0 => 011, 110
- Step 1: 110 => M = -A = 0000 0100
- Step 2: 011 =>
  M = M + 4*(2A) = 0000 0100 + 11100000
  = 1110 0100 = -28 (dec)

Wallace-Tree



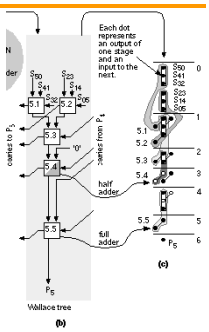
Improving Speed

- Collapse the chain of FAs a0-f5 (5 adders delays) to the Wallace tree consisting of 5.1-5.4 (4 adders delays)
- To form P_5 use
  - Summands: S_{50}, S_{41}, S_{32}, S_{23}, S_{14}, S_{05}
  - 4 carries from P_4



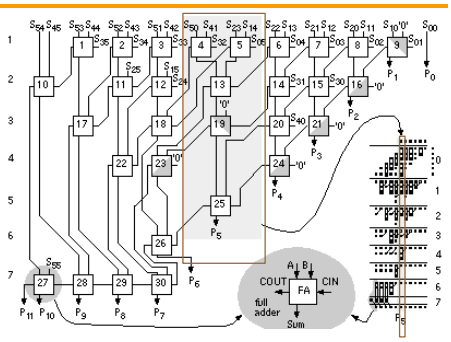
What is Game?

- Dots and holes –
  the outputs of one stage = inputs of the next
- At each stage we have three choices
  (1) sum 3 outputs using Full Adder –
  box with 3 dots
- (2) sum 2 outputs using Half Adder –
  box with 2 dots
- (3) pass outputs directly to the next stage
- Choose (1), (2), or (3) at each stage to maximize the performance of the multiplier
- Tree-based multipliers
  - Work Forward (Wallace-tree Multiplier)
  - Work Backward (Dadda Multiplier)



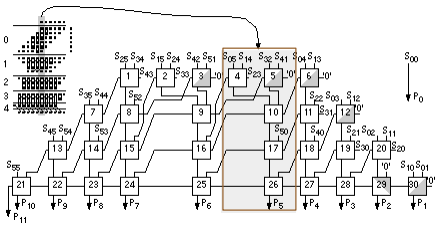
6-bit Wallace Multiplier

- Complexity
  CSA – 26 (incl. 6 HAs)
  CPA – 4
- Delay:
  CSA – 6 adders delay
  + CPA – 4



6-bit Dadda Multiplier

- Complexity
  CSA – 20 (incl. 4 HAs)
  CPA – 10
- Delay:
  CSA – 3 adders delay
  + CPA delay



Work Backward:
each successive stage is 3/2 times larger

ARM Multiplier design

- All ARMs apart form the first prototype have included support for integer multiplication
  - older ARM cores include low-cost multiplication hardware that supports only the 32-bit result multiply and multiply-accumulate
  - recent ARM cores have high-performance multiplication hardware and support 64-bit result multiply and multiply-accumulate
- Low cost implementation
  - Use the datapath iteratively, employing the barrel shifter and ALU to generate 2-bit product in each clock cycle
  - use early termination to stop the iterations when there are no more ones in the multiply register

The 2-bit multiplication algorithm, Nth cycle

- Control settings for the Nth cycle of the multiplication
- Use existing shifter and ALU + additional hardware
  - dedicated two-bits-per-cycle shift register for the multiplier and a few gates for the Booth's algorithm control logic (overhead is a few per cent on the area of ARM core)

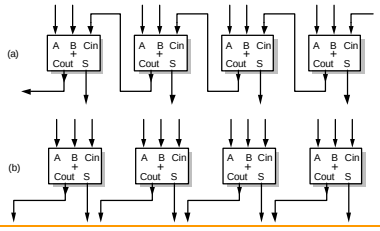
Carry-in	Multiplier	Shift	ALU	Carry-out
0	x 0	LSL #2N	A + 0	0
	x 1	LSL #2N	A + B	0
	x 2	LSL #(2N + 1)	A – B	1
	x 3	LSL #2N	A – B	1
1	x 0	LSL #2N	A + B	0
	x 1	LSL #(2N + 1)	A + B	0
	x 2	LSL #2N	A – B	1
	x 3	LSL #2N	A + 0	1

High speed multiplication

- Where multiplication performance is very important, more hardware resources must be dedicated
  - in some embedded systems the ARM core is used to perform real-time digital signal processing (DSP) – DSP programs are typically multiplication intensive
- Use intermediate results which include partial sums and partial carries
  - Carry-save adders are used for this
- These two binary results are added together at the end of multiplication
  - The main ALU is used for this

Carry-propagate (a) and carry-save (b) adder structures

- Carry propagate adder takes two conventional (irredundant) binary numbers as inputs and produces a binary sum
- Carry save adder takes one binary and one redundant (partial sum and partial carry) input and produces a sum in redundant binary representation (sum and carry)



ARM high-speed multiplier organization

- CSA has 4 layers of adders each handling 2 multiplier bits => multiply 8-bits per clock cycle
- Partial sum and carry are cleared at the beginning or initialized to accumulate a value
- Multiplier is shifted right 8-bits per cycle in the 'Rs' register
- Carry sum and carry are rotated right 8 bits per cycle
- Performance: up to 4 clock cycles (early termination is possible)
- Complexity: 160 bits in shift registers, 128 bits of carry-save adder logic (up to 10% of simpler cores)

ARM high-speed multiplier organization