CPSC 3615 Computer Architecture

Chapter 10

Adders and Simple ALUs
(B. Parhami's Text Book: Chapter 10)

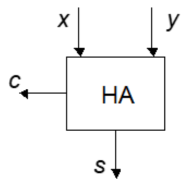
(notes revised from the resources of the textbook)

10.1 Simple Adders

. Half adder: 2 inputs

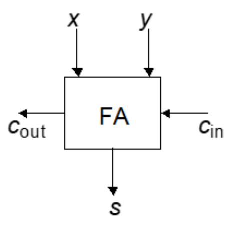
c=x AND y (carry)
s= x XOR y (sum)

Inputs		Outputs	
x	*y*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



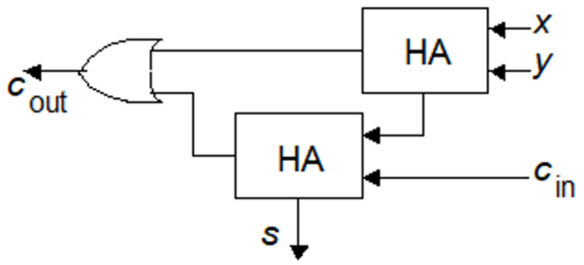
. Full adder: 3 inputs

Inputs			Outputs	
x	*y*	c_{in}	c_{out}	*s*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Binary half-adder (HA) and full-adder (FA).

Full-Adder Implementations



FA built of two HAs

Full adder implemented with two half-adders,

.Ripple-Carry Adder: Slow But Simple

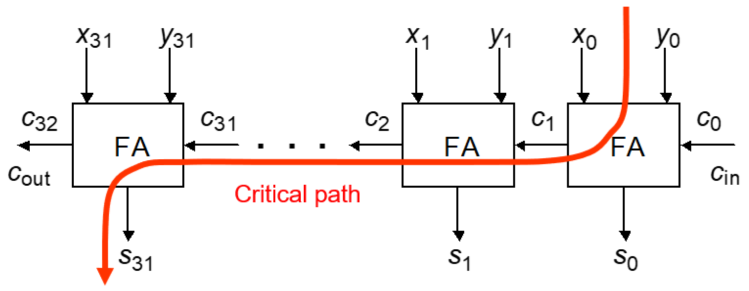


Figure 10.4 Ripple-carry binary adder with 32-bit inputs and output.

10.2 Carry Propagation Networks

To design the fast adder, the delay of carry generations should be avoided. We introduce a carry propagation networks.

eg. x+y

First, define $g_i = x_i\ y_i$ (logical and), $p_i = x_i$ XOR y_i

if $g_i = 1$, a new carry is generated

$x_i = 1$, $y_i = 1$

$$\begin{array}{r} 1 \\ +\ 1 \\ \hline =\ 1\,0 \end{array}$$

↰ a new carry is generated

$x_i = 0$, $y_i = 1$

$$\begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array}$$

No Carry

So there are two cases for a carry to the next bit:

1. a new carry is generated $g_i=1$, or
2. an input carry is propagated to the next bit $p_i=1$ and $c_i=1$

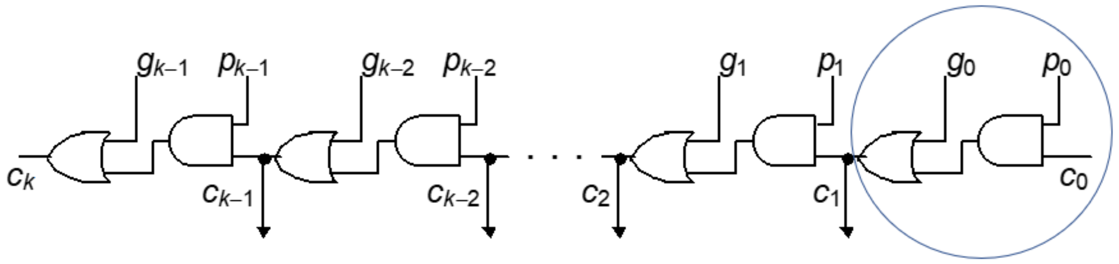
That is: $C_{i+1}= g_i \vee p_i c_i$

Ripple-Carry Adder Revisited

The carry recurrence: $c_{i+1} = g_i \vee p_i c_i$

Latency of *k*-bit adder is roughly 2*k* gate delays:

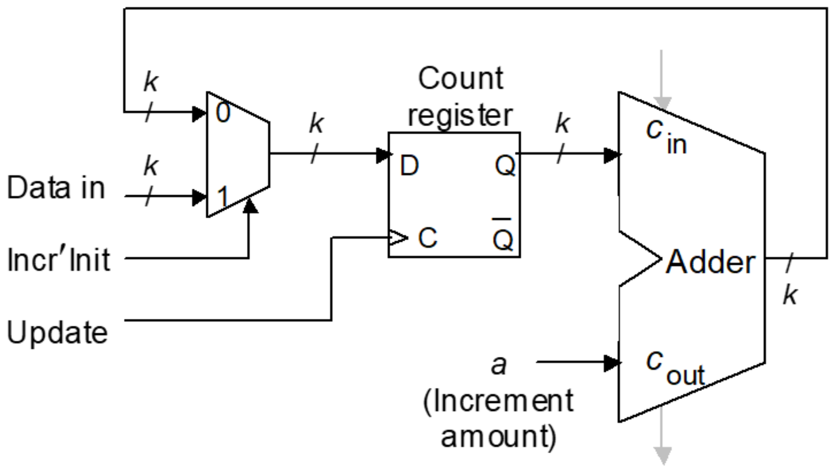
1 gate delay for production of *p* and *g* signals, plus
2*k* gate delays for carry propagation, plus
1 XOR gate delay for generation of the sum bits



The carry propagation network of a ripple-carry adder.

10.3 Counting and Incrementation

Increase by a,   x=x+a



Schematic diagram of an initializable synchronous counter.

10.4 Design of Fast Adders

carries can be computed directly without delay

In the carry-lookahead adders, the i-th carry is obtained from the input of g_j, p_j, C_0 ($0 <= j <= i-1$)

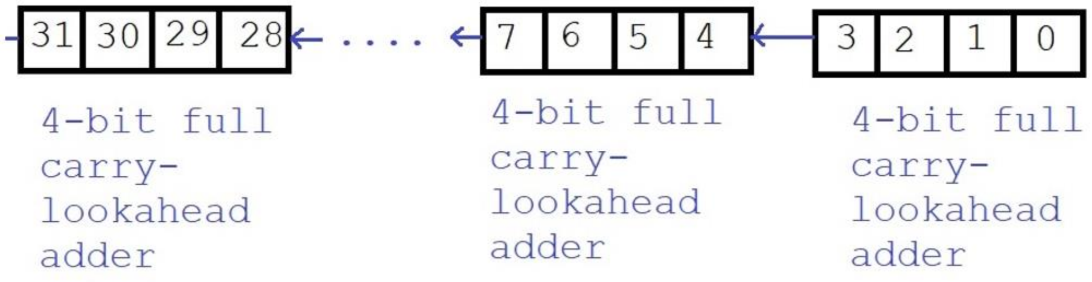
$$C_1 = G_0 + P_0 \cdot C_0,$$
$$C_2 = G_1 + P_1 \cdot C_1,$$
$$C_3 = G_2 + P_2 \cdot C_2,$$
$$C_4 = G_3 + P_3 \cdot C_3.$$

Substituting C_1 into C_2, then C_2 into C_3, then C_3 into C_4 yields the following expanded equations:

$$C_1 = G_0 + P_0 \cdot C_0,$$
$$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1,$$
$$C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2,$$
$$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3.$$

When designing 32-bit fast adder, in order to make it simpler, we separate the 32-bit into 8 groups, each group has 4 bits.

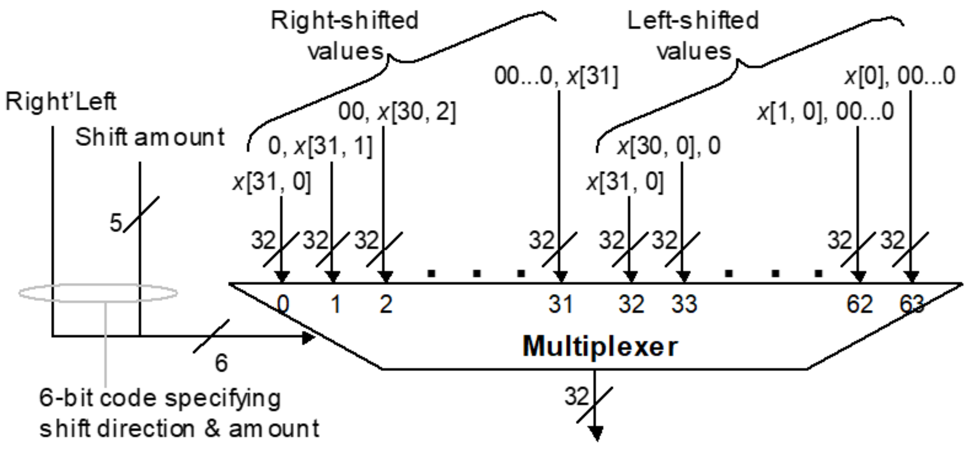
Inside each group, we use 4-bit full carry-lookahead adder. Between the groups, the structure of ripple-carry adder are applied.


31 30 29 28
7 6 5 4
3 2 1 0
4-bit full carry-lookahead adder
4-bit full carry-lookahead adder
4-bit full carry-lookahead adder

10.5 Logic and Shift Operations

Logic and, or, nor, xor can be implemented by the corresponding gates.

Shifts can be implemented by multiplexing.



Multiplexer-based logical shifting unit.

Arithmetic Shifts

Purpose: Multiplication and division by powers of 2

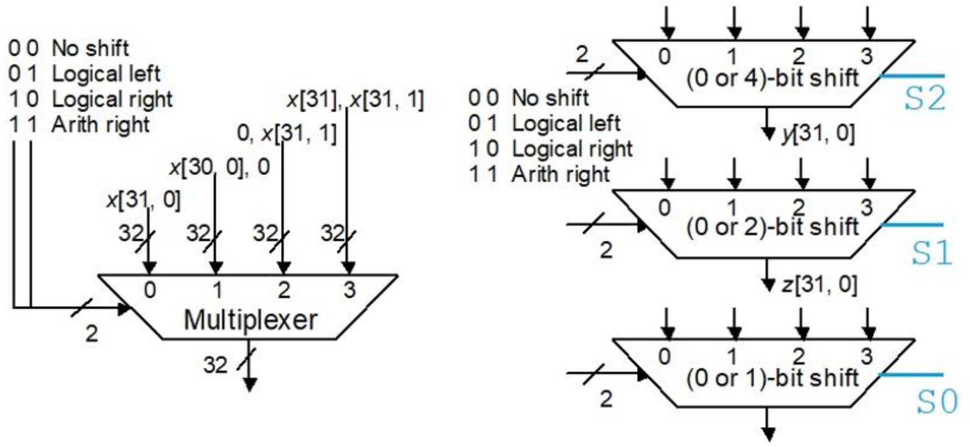
```
sra  $t0,$s1,2 # $t0 ← ($s1) right-shifted by 2
srav $t0,$s1,$s0  # $t0 ← ($s1) right-shifted by ($s0)
```

	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 … 0
R	0 0 0 0 0 0	0 0 0 0 0	1 0 0 0 1	0 1 0 0 0	0 0 0 1 0	0 0 0 0 1 1
	ALU instruction	Unused	Source register	Destination register	Shift amount	sra = 3

	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 … 0
R	0 0 0 0 0 0	1 0 0 0 0	1 0 0 0 1	0 1 0 0 0	0 0 0 0 0	0 0 0 1 1 1
	ALU instruction	Amount register	Source register	Destination register	Unused	srav = 7

The two arithmetic shift instructions

Shifting in Multiple Stages



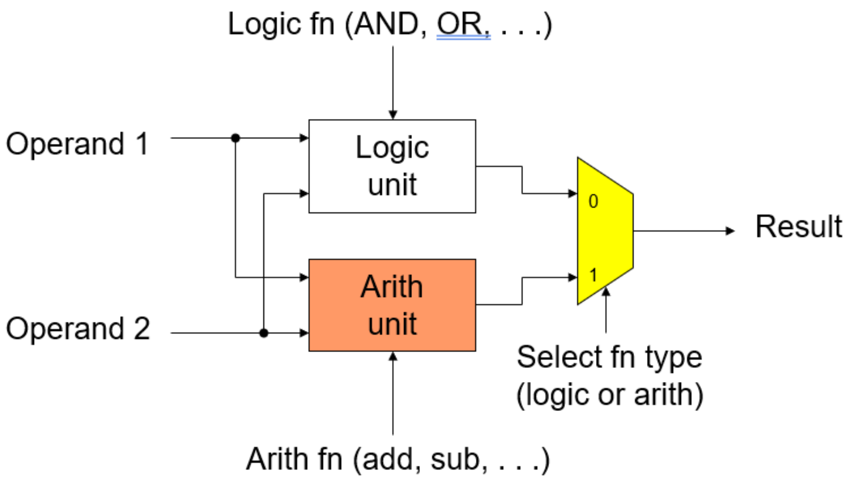
(a) Single-bit shifter

(b) Shifting by up to 7 bits

S2 S1 S0: 1 0 1 shift 5 bits		1 1 1 shift 7 bits
0 1 1 shift 3 bits		1 0 0 shift 4 bits

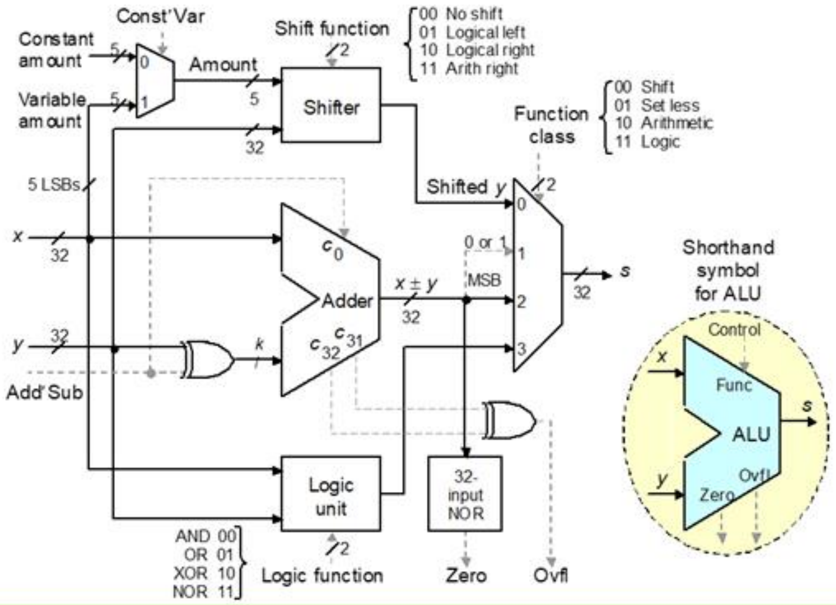
Multistage shifting in a barrel shifter.

10.6 Multifunction ALUs



Logic fn (AND, OR, . . .)
Operand 1
Logic unit
Arith unit
Operand 2
0
1
Result
Select fn type (logic or arith)
Arith fn (add, sub, . . .)

General structure of a simple arithmetic/logic unit.



A multifunction ALU with 8 control signals (2 for function class, 1 arithmetic, 3 shift, 2 logic) specifying the operation.