04s2 COMP3211/9211 Computer Architecture
Tutorial 2 Solutions

Lih Wen Koh (lwkoh@cse.unsw.edu.au)
05 September 2004

[Key: SRQ = Stallings, Review Question; SP = Stallings Problem; P = Patterson & Hennessy Exercise;Y = Yalamanchili]

Hand-in Exercises

Q1. Given two integers, m and n, an m x n integer array A, and an n x 1 vector b stored in memory, design a high-level program fragment to compute and store back to memory the m x 1 vector product r = A x b.

Translate your program into an efficient MIPS assembly language implementation of it. Do not optimise the instruction schedule to eliminate any data dependencies or branch delays that may be present in your code (don't worry if you don't yet know what these are: your program will then be what is desired by default).

Describe which registers you have allocated to storing the local variables and memory addresses used by your program.

Sketch a memory map that indicates where your program and data will reside in memory assuming each will occupy contiguous addresses in a single unified memory block.

High Level Program Fragment

/* Given integers ***m*** and ***n***, assume that the integer arrays ***A*** (*mxn*) and ***b*** (*nx1*) are initialized. The *mx1* integer array ***r*** stores the matrix multiplication result of A and b. */

```
int m, n;
int A[m][n], b[n], r[m];
```

/* local variables used for program calculation */

```
int sop, i, j;
```

/* Matrix multiplication */

```
for i from 0 to (m-1) do
      sop = 0;
      for j from 0 to (n-1) do
           sop += A[i][j] * b[j];
      end for;
      r[i] = sop;
end for;
```

return r ;

MIPS Assembly Language Translation of Program

Assumptions:

- The memory is large enough (at least 108 + 4(m*n + n + m) bytes) to contain the program, data and perhaps some stack space.
- Integers are 4-byte data in the MIPS architecture.
- The elements of integer arrays *A*, *b* and *r* are stored in contiguous memory.
- The start address of the data segment in memory is known and is stored in register $t0.
- The base addresses of *A*, *b* and *r* are known and are stored in registers $t1, $t2, $t3 respectively.
- The integer product of *A*[i][j] and *b*[j] fits into a 32-bit value, i.e. no multiplication overflow will occur. (Multiplication overflow flag must be checked if this assumption is not considered)
- Program startup and linking sections are not included in this design.

The following registers are used:

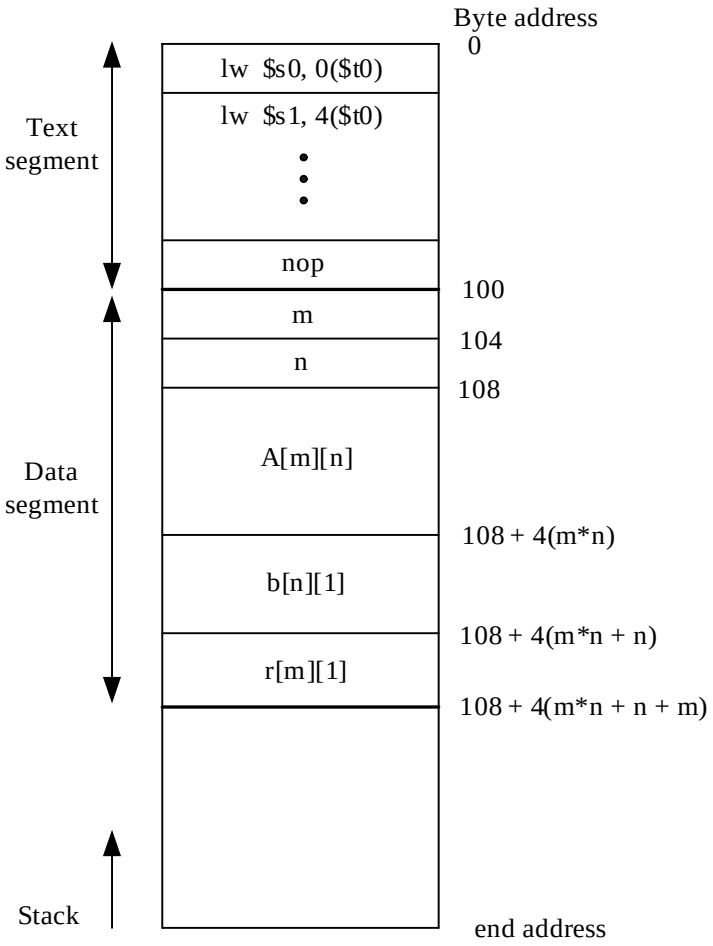
$zero always has the value 0.
$s0 as ***m***,
$s1 as ***n***,
$s2 as ***i***,
$s3 as ***j***,
$s4 as ***sop***,
$s5 as the element ***A[i][j]***,
$s6 as the element ***b[j]***,
$s7 as the multiplication result ***A[i][j]*b[j]***,
$t0 contains the start address of the data segment in memory,
$t1 contains the base address for matrix *A*,
$t2 contains the base address for matrix *b*,
$t3 contains the base address for matrix *r*,
$t4 as the *offset* to the base address of matrix *A*,
$t5 as the *offset* to the base address of matrix *b*,
$t6 as the *offset* to the base address of matrix *r*,
$t7 as the addition of a base address and an offset.

```
(1)             lw      $s0,   0($t0)          | Load m into $s0 ;
(2)             lw      $s1,   4($t0)          | Load n into $s1 ;
(3)             add     $s2,   $zero,  $zero   | Clear i ;
(4)             add     $t4,   $zero,  $zero   | Point to first element of A ;
(5)             add     $t5,   $zero,  $zero   | Point to first element of b ;
(6)             add     $t6,   $zero,  $zero   | Point to first element of r ;
        for1:
(7)             add     $s4,   $zero,  $zero   | Clear sop ;
(8)             add     $s3,   $zero,  $zero   | Clear j ;
        for2:
(9)             add     $t7,   $t1,    $t4     |
(10)            lw      $s5,   0($t7)          | Load A[i][j] ;
(11)            add     $t7,   $t2,    $t5     |
(12)            lw      $s6,   0($t7)          | Load b[j] ;
(13)            mult    $s5,   $s6             | A[i][j] * b[j] ;
(14)            mflo    $s7                    | Move 32-bit integer product into $s7;
                                               | Multiplication overflow is not checked here.
(15)            add     $s4,   $s4,    $s7     | sop += A[i][j] * b[j] ;
(16)            addi    $s3,   $s3,    1       |  j++ ;
```

```
(17)            bne     $s3,  $s1,  for2      | Conditional branch to label for2 ;
        endfor2:
(18)            add     $t7,  $t3,  $t6       |
(19)            sw      $s4,  0($t7)          | r[i] = sop ;
(20)            addi    $t4,  $t4,  4         | Point to next element of A ;
(21)            addi    $t5,  $t5,  4         | Point to next element of b ;
(22)            addi    $t6,  $t6,  4         | Point to next element of r ;
(23)            addi    $s2,  $s2,  1         | i++ ;
(24)            bne     $s2,  $s0,  for1      | Conditional branch to label for1 ;
        endfor1:
(25)            nop                           | End of program;
```

We may use the ***nop*** (No Operation) instruction to indicate the end of the program fragment. The control logic will de-assert all control signals on encountering a *nop* instruction. Alternatively, we could also use the instruction "beq $zero, $zero, endfor1" to keep the PC looping at the label *endfor1* so that the PC will never enter the data segment in memory.

Memory Map for Program and Data



Q2. [P4.51] Calculate the delays to add four 16-bit numbers using full carry-lookahead adders versus carry save with a carry-lookahead adder forming the final sum.

The simplified version of the problem is given in P&H Problem 4.50:
Assume that the time delay through each 1-bit adder is 2T. Calculate the time of adding four 4-bit numbers to the organization at the top versus the organization in the bottom in Figure 4.56.

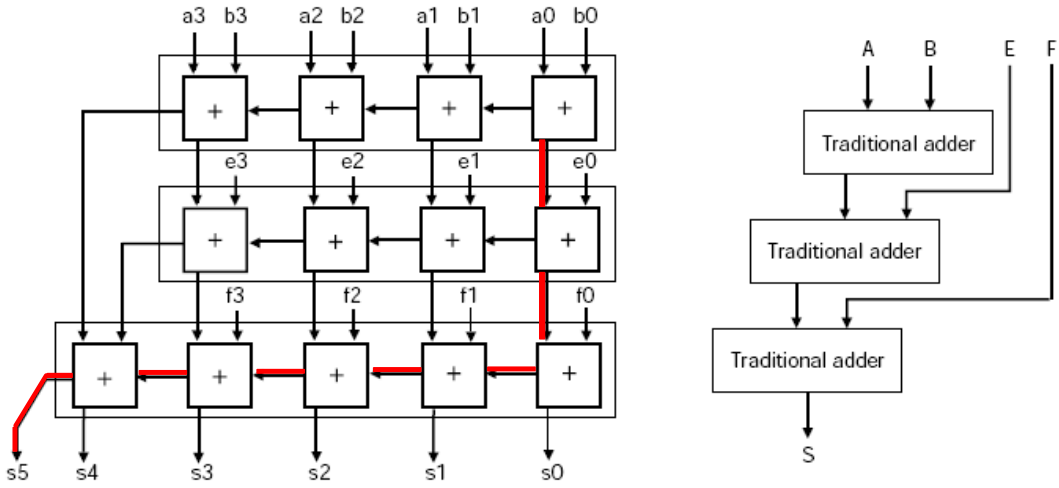


Figure 1: Adding four 4-bit numbers using ripple carry adders.

The critical path for the adder in Figure 1 is highlighted in red, with a propagation delay of **14T**.

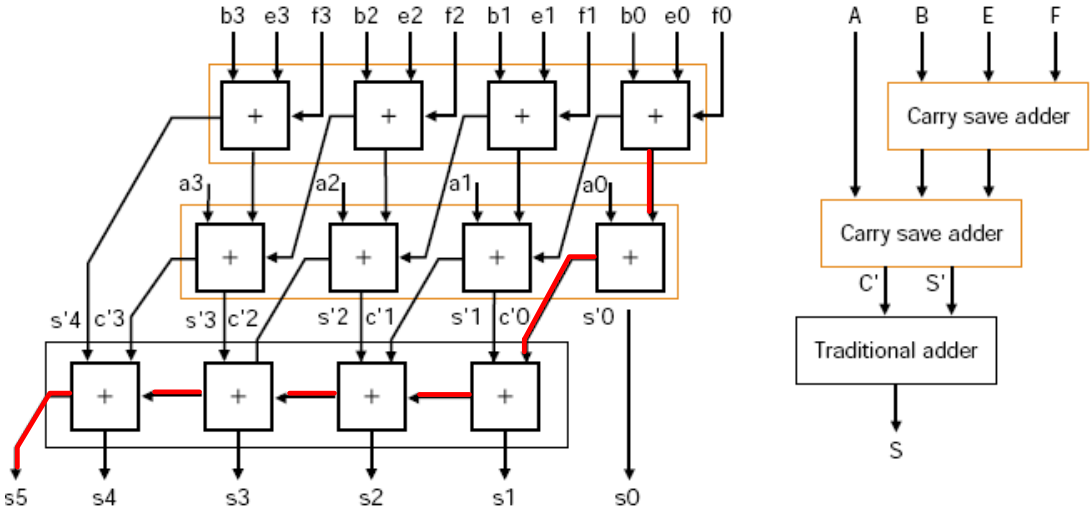


Figure 2: Adding four 4-bit numbers using carry save adders with a ripple carry adder forming the final sum:

The critical path for the carry-save adder is highlighted in red, with a propagation delay of **12T**.

Now, we come back to the problem of adding four 16-bit numbers using full carry-lookahead adders. Using a similar structure to Figure 1, we replace the 4-bit ripple carry adders in Figure 1 by 16-bit carry-lookahead adders. Please refer to Q7 for the construction of a 16-bit full carry lookahead adder and the analysis of propagation delays for carry lookahead adders.

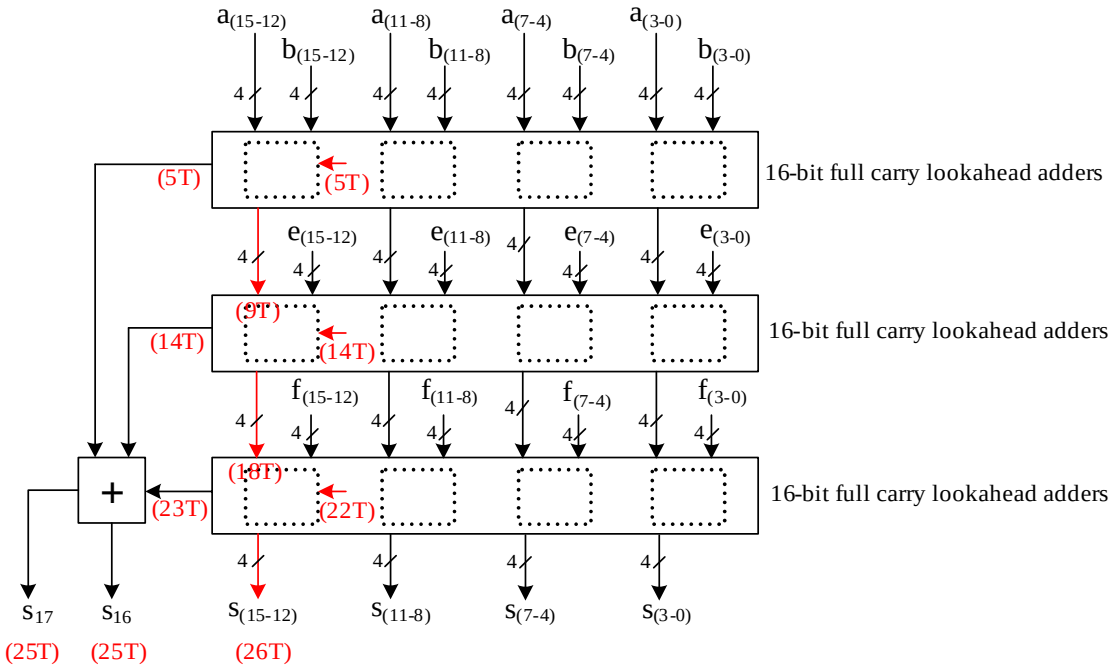


*Figure 3: The critical path for the full carry-lookahead adder is highlighted in red, with a propagation delay of **26T**.*

Adding four 16-bit numbers using carry save adders with a carry-lookahead adder forming the final sum:

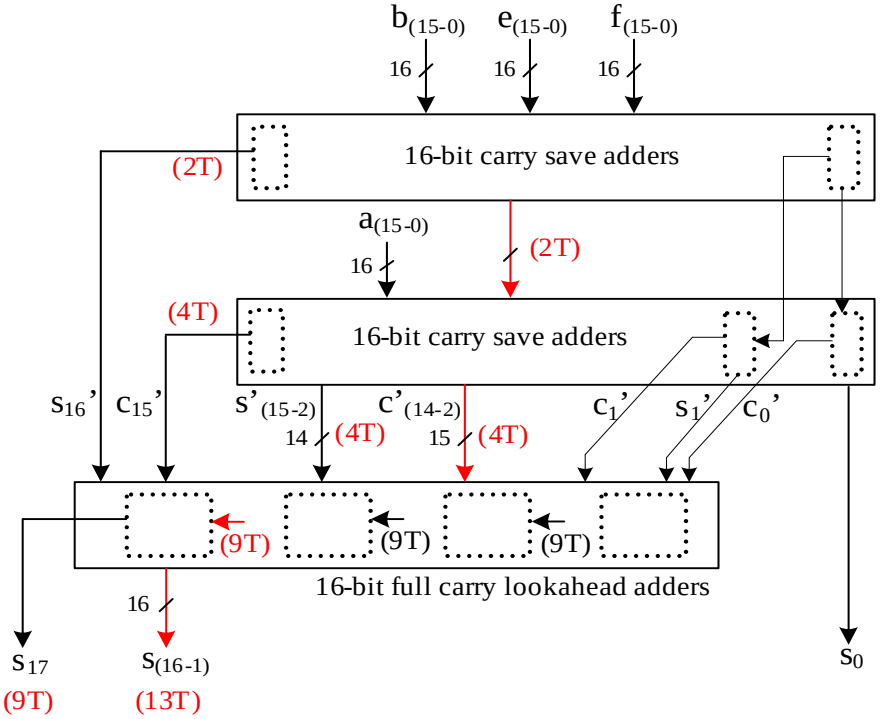


*Figure 4: The critical path for carry save adders with a carry-lookahead adder forming the final sum is highlighted in red, with a propagation delay of **13T**.*

Warmup Exercises

Q3. [P4.10] Find the shortest sequence of MIPS instructions to determine the absolute value of a two's complement integer. Convert this instruction (accepted by the MIPS assembler):

abs $t2, $t3

Solution

```
        addu   $t2, $zero, $t3     | copy $t3 into $t2
        bgez   $t3, next           | if $t3 >= 0 then done
        sub    $t2, $zero, $t3     | else $t2 = -$t3
next:
```

Q4. [P4.14] Given the bit pattern: 1000 1111 1110 1111 1100 0000 0000 0000; what does it represent, assuming that it is

a. a two's complement integer?
b. an unsigned integer?
c. a single precision floating-point number?
d. a MIPS instruction?

Solution

a. The sign bit is 1, so this is a negative number. We first take its two's complement.

A = 1000 1111 1110 1111 1100 0000 0000 0000
$-A$ = 0111 0000 0001 0000 0100 0000 0000 0000
$= 2^{30} + 2^{29} + 2^{28} + 2^{20} + 2^{14}$
= 1,073,741,824 + 536,870,912 + 268,435,456 + 1,048,576 + 16,384
= 1,880,113,152

Thus A = –1,880,113,152

b. A = 1000 1111 1110 1111 1100 0000 0000 0000
= 8*FEF*C000
$= 8 * 16^7 + 15 * 16^6 + 14 * 16^5 + 15 * 16^4 + 12 * 16^3$
= 2,147,483,648 + 251,658,240 + 14,680,064 + 983,040 + 49,152
= 2,414,854,144

c. (Please refer to P&H textbook page 276 for the format of single precision floating-point numbers)

s = 1
exponent = 0001 1111
= 25 – 1
= 31
significand = 110 1111 1100 0000 0000 0000

$(-1)^S * (1 + \text{significand}) * 2^{\text{exponent}-127} = -1 * 1.1101\ 1111\ 1 * 2^{-96}$
$= -1 * (1 + 13 * 16^{-1} + 15 * 16^{-2} + 2^{-9}) * 2^{-96}$
$= -1.873 * 2^{-96}$
$= -2.364 * 10^{-29}$

d. opcode (6 bits) = 100011 = lw
rs (5 bits) = 11111 = 31
rt (5 bits) = 01111 = 15

address (16 bits) = 1100 0000 0000 0000
Since the address is negative we have to take its two's complement.
Two's complement of address = 0100 0000 0000 0000
Address = -2^{14} = –16384

Therefore the instruction is lw 15, –16384(31).

Notice that the address embedded within the 16-bit immediate field is a byte address unlike the constants embedded in PC-relative branch instructions where word addressing is used.

Q5. [P4.15] Similar to Q4, but this time use the bit pattern 0000 0000 0000 0000 0000 0000 0000 0000.

a. 0
b. 0
c. 0.0
d. sll $0, $0, 0

Discussion Questions

Q7. [P4.46] 4-bit Ripple Carry Adder vs. Carry Lookahead Adder

Let's start with a 1-bit full adder with inputs a_i, b_i, c_i and outputs sum s_i and carry out c_{i+1}.

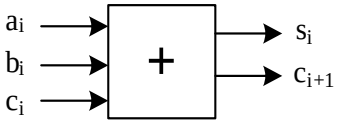


Figure 5: 1-bit full adder

Using Karnaugh map, we can obtain the equations of sum s_i and carry out c_{i+1} in Sum-of-Product (SOP) form or Product-of-Sum (POS) form. Equations expressed as either SOP or POS can be easily implemented as two-level circuits assuming that both complemented and uncomplemented inputs are available. Thus we assume that the propagation delay for a 1-bit full adder is **2T**, where T is the propagation delay through one level of gate(s).

A 4-bit ripple carry adder is constructed using four 1-bit full adders where the carry-out of the less significant bit adder is chained to the carry-in of the next more significant bit adder.

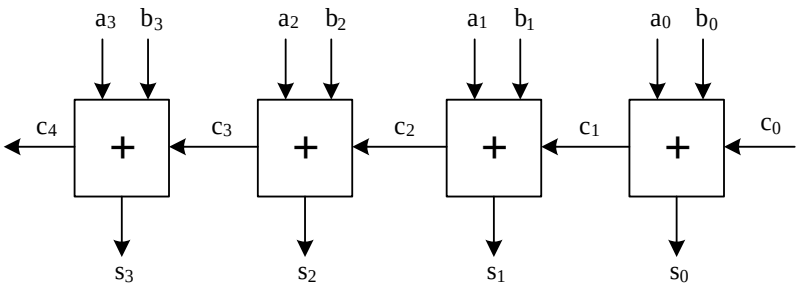


Figure 6: 4-bit ripple carry adder

The propagation delay of the most significant sum bit, s_3 , of a 4-bit ripple carry adder
= 2T for c_1 + 2T for c_2 + 2T for c_3 + 2T for two-level gates for s_3
= **8T**

The propagation delay of the carry out bit, c_4 , of a 4-bit ripple carry adder
= 2T for c_1 + 2T for c_2 + 2T for c_3 + 2T for c_4
= **8T**

In order to construct a carry lookahead adder, we define two additional output signals for a 1-bit full adder:

Propagate signal: $p_i = a_i + b_i$
Generate signal: $g_i = a_i \bullet b_i$

each of which has propagation delay **T.**

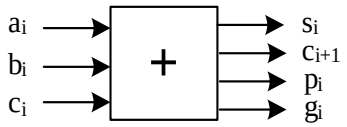


Figure 7: 1-bit full adder with propagate and generate outputs.

We can re-write the carry out signal c_i in terms of p_i and g_i such that

$$\begin{aligned} c_{i+1} &= a_i b_i + a_i c_i + b_i c_i \\ &= a_i b_i + c_i (a_i + b_i) \\ &= g_i + p_i c_i \end{aligned}$$

Thus we can write the carry-out signal c_i for each bit adder as:

$$\begin{aligned} c_1 &= g_0 + p_0 c_0 \\ c_2 &= g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0 \\ c_3 &= g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\ c_4 &= g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \end{aligned}$$

The idea is to directly generate the carry-out signal c_o for each bit adder without having to wait for the chained carry-in signal c_i.

The propagation delay of the carry out bit, c_1 , of a 4-bit carry lookahead adder
= max(T for g_0, T for p_0, zero delay for c_0) + 2T for two-level gates for c_1
= **3T**

Similarly, the propagation delays of the carry out bits, c_2 , c_3 and c_4, of a 4-bit carry lookahead adder
= **3T**

The propagation delay for the most significant sum bit, s_3 , of a 4-bit carry lookahead adder
= 3T for c_3 + 2T for two-level gates for s_3
= **5T**

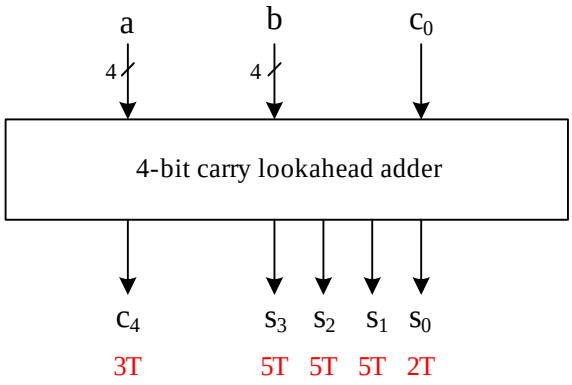


Figure 8: 4-bit carry lookahead adder (with propagation delay for each output signal highlighted in red).

Extra: To construct a 16-bit carry lookahead adder using four 4-bit carry lookahead adders, we use the same principles as before i.e. compute the carry-out signal of each 4-bit carry lookahead adder using *propagate* and *generate* signals (refer to Figure 8).

We define

$$P_{i-(i+3)} = p_{(i+3)}\,p_{(i+2)}\,p_{(i+1)}\,p_i$$
$$G_{i-(i+3)} = g_{(i+3)} + p_{(i+3)}\,g_{(i+2)} + p_{(i+3)}\,p_{(i+2)}\,g_{(i+1)} + p_{(i+3)}\,p_{(i+2)}\,p_{(i+1)}\,g_i$$

such that

$$P_{0-3} = p_3\,p_2\,p_1\,p_0$$
$$G_{0-3} = g_3 + p_3\,g_2 + p_3\,p_2\,g_1 + p_3\,p_2\,p_1\,g_0 \quad \text{etc.}$$

Then

$$c_4 = G_{0-3} + P_{0-3}\,c_0$$
$$c_8 = G_{4-7} + P_{4-7}\,c_4$$
$$= G_{4-7} + P_{4-7}\,G_{0-3} + P_{4-7}\,P_{0-3}\,c_0$$
$$c_{12} = G_{8-11} + P_{8-11}\,c_8$$
$$= G_{8-11} + P_{8-11}\,G_{4-7} + P_{8-11}\,P_{4-7}\,G_{0-3} + P_{8-11}\,P_{4-7}\,P_{0-3}\,c_0$$
$$c_{16} = G_{12-15} + P_{12-15}\,c_{12}$$
$$= G_{12-15} + P_{12-15}\,G_{8-11} + P_{12-15}\,P_{8-11}\,G_{4-7} + P_{12-15}\,P_{8-11}\,P_{4-7}\,G_{0-3} + P_{12-15}\,P_{8-11}\,P_{4-7}\,P_{0-3}\,c_0$$

Thus

The propagation delay of the carry-out bit, c_4 , of a 16-bit full carry lookahead adder
= max(3T for G_{0-3}, 2T for P_{0-3}, zero delay for c_0) + T for the equation for c_4
= **4T**

The propagation delay of the carry-out bit, c_7 , of a 16-bit full carry lookahead adder
= max(T for *p's* and *g's*, 4T for c_4) + 2T for two-level circuits for c_7
= **6T**

The propagation delay of the sum bit, s_7 , of a 16-bit full carry lookahead adder
= max(zero delay for a_7, zero delay for b_7, 6T for c_7) + 2T for two-level circuits for s_7
= **8T**

etc etc…

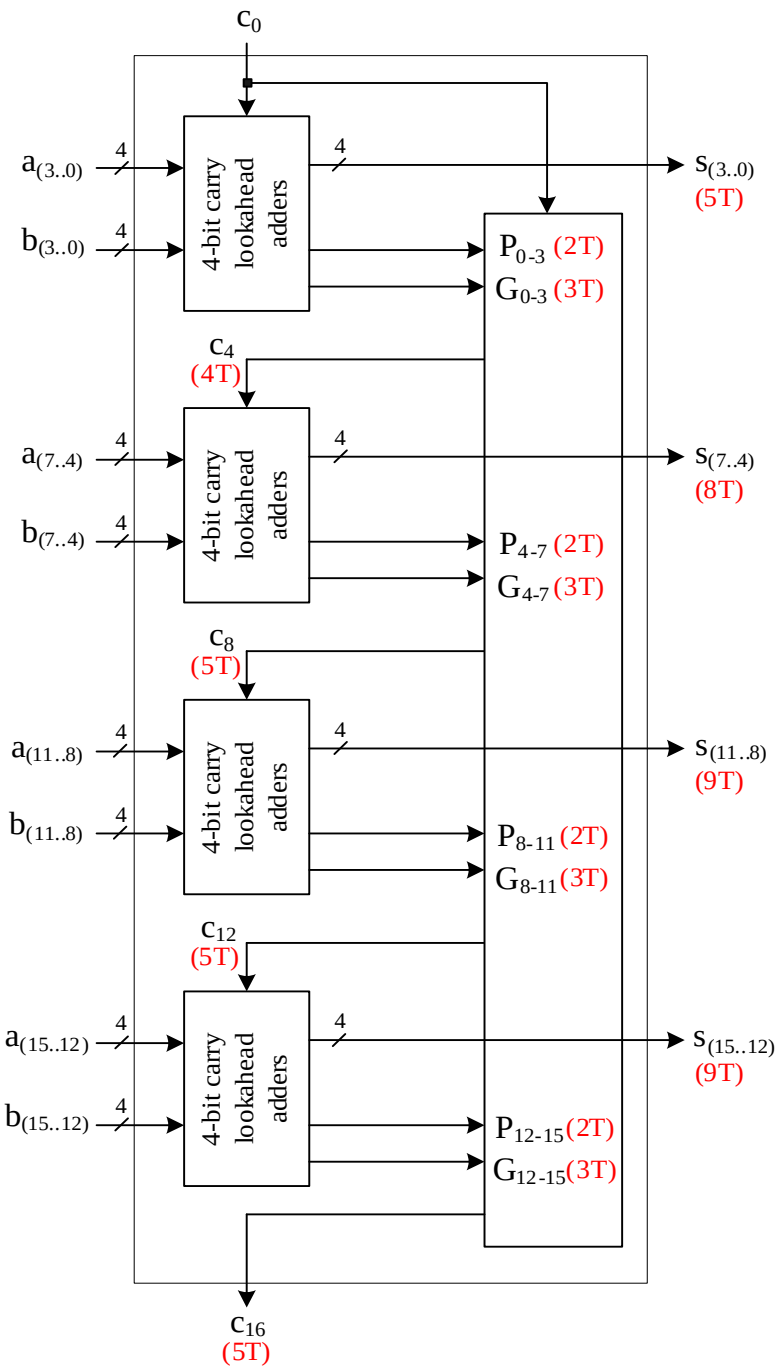


Figure 9: 16-bit full carry lookahead adder (with propagation delay for each output signal highlighted in red).

Q8. [Y5.3] You are part of a software group developing algorithms for processing speech signals for a new digital signal processing chip. To test your software your options are to construct

i) a detailed hierarchical model of the chip comprised of gate level models at the lowest level of the hierarchy; or
ii) a behavioural level model of the chip that can implement the algorithms that you wish to use.

Your goal is to produce correct code for a number of algorithms prior to detailed testing on a hardware prototype. How would you evaluate these choices and what are the trade-offs in picking one approach over the other?

(+: advantage, –: disadvantage)

Hierarchical model:
+ Building blocks can be heavily optimized.
+ Reuse of verified building blocks.
+ Performance/delay/power analysis can be performed in stages (building blocks).
– Requires knowledge & experience in hardware design.

Behavioural model:
+ Design simplicity & speed.
– Performance/power analysis may be difficult due to the abstraction of hardware blocks.

The behavioural model is preferred if the goal is to rapidly produce correct code for prototyping purpose.