SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

ISTD 50.002 Computation Structures
2D Project

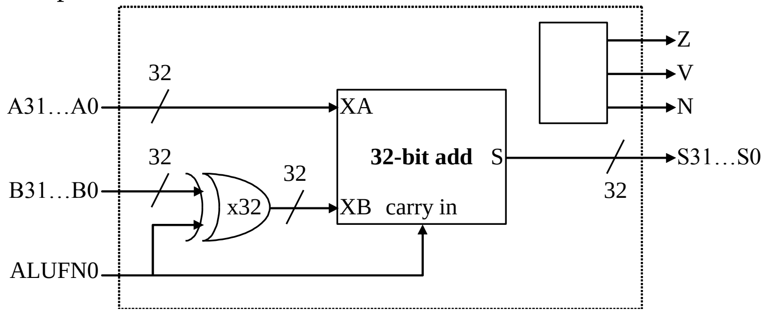
Part 1: 32-bit Adder Optimization

Instructions

1. Design and debug your circuit as described below
2. Using the appropriate checkoff file, verify that your circuit operates correctly and submit it to the on-line checkoff system.
3. Demo to your instructors.

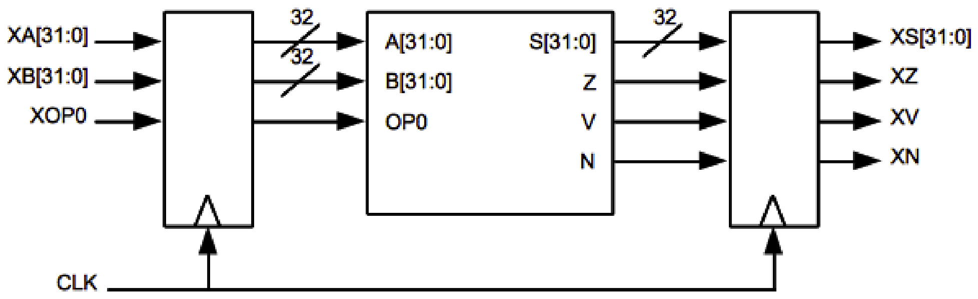
Project Description

The goal of this design project is to improve the performance of the 32-bit adder/subtractor unit you built as part of Lab #3:



A detailed description of the desired functionality can be found in Lab #3 and your optimized design should meet all the functional requirements set forth there.

The test circuitry supplied by the checkoff file inserts the adder/subtractor unit into a pipeline stage and tests the circuit with one of three clock periods to determine the unit's performance:



Note that the register has a t_{PD} = 190 ps + load-dependent-delay, so the load your circuit places on the inputs will affect the time at which the unit's input signals become valid after the rising edge of the clock.

A unit implemented using a ripple-carry adder built from XOR, AND, and OR gates can be clocked with a period of approximately 14 ns. The amount of credit you'll receive for your design depends on its performance level:

Credit received	t_{CLK}	*Checkoff file*
0 points	100 ns	/50002/2dcheckoff_100ns.jsim
2 points	6 ns	/50002/2dcheckoff_6ns.jsim
5 points	4 ns	/50002/2dcheckoff_4ns.jsim
10 points	3 ns	/50002/2dcheckoff_3ns.jsim

The checkoff files are identical except for the clock period used to test your circuit. The 0-point file isn't intended for checkoff but may be useful when debugging your circuit. The on-line system determines the appropriate number of points to record from the checkoff file you use; you can resubmit your design more than once if you continue to improve its performance and can use a faster checkoff file.

Since the test jig contains registers, JSim will report the "minimum observed setup time" at the end of each simulation run. At each rising clock edge, JSim determines the setup time for each data input to a register (i.e., how long the data inputs were valid before the rising clock edge). JSim remembers the smallest setup time it finds, along with the simulated time it made the observation and the device involved. So, for example, JSim might report

```
min observed setup = 85.951ns @ time=4.3us (device = xz.x1)
```

The reported device is the register used to capture the value of the Z bit from the adder/subtractor unit. Assuming t_{CLK} = 100 ns, this would suggest that the critical path through your circuit is 14.049 ns and involves the generation of the Z signal during clock cycle 43.

You should use the following .subckt definition for your adder/subtractor unit so that it can be tested by 2dcheckoff_XX.jsim:

```
.include '/50002/nominal.jsim'
.include '/50002/stdcell.jsim'
.include '/50002/2dcheckoff_XX.jsim'
```

```
.subckt adder32 op0 A[31:0] B[31:0] S[31:0] z v n
... your circuit here ...
.ends
```

where XX is one of "100ns", "6ns", "4ns", or "3ns". Op0 will be set to 0 for ADD (S = A + B) and 1 for a SUBTRACT (S = A – B); A[31:0] and B[31:0] are the 32-bit two's complement input operands; S[31:0] is the 32-bit result; z/v/n are the three condition code bits described in Lab #3.

Design Hints

Here are some ideas on where to start when trying to optimize circuit performance:

1. *Minimize load-dependent delays*. As you connect the output of a gate to additional inputs, its capacitance increases and so changes in the signal value take longer (see t_R and t_F columns in the standard cell table of Lab #3). Heavily loaded signals should be buffered to reduce the total delay. There are several strengths of INVERTERs and BUFFERs available for driving different sized loads.

2. *Use inverting logic*. NAND, NOR, etc. gates are noticeably faster than their non-inverting counterparts(AND, OR, …). Rearranging logic to use inverting gates can make a big difference performance along the critical path of your circuit. Positive logic gates are included in the standard cell library because they are often smaller than their inverting logic equivalents and thus may be useful for implementing logic that is not on the critical path.

3. *Minimize delay along critical paths*. It's often possible to rearrange your logic to reduce the number of gate delays along the critical path, sometimes at the cost of additional gates elsewhere but that's a tradeoff which is often worthwhile. In a ripple-carry adder, you want to minimize the CIN to COUT delay when trying to optimize overall circuit performance.

4. *Use different adder architecture*. Consider using an architecture that avoids rippling the carry through all 32 bits of the adder: carry-select and carry-lookahead are two techniques. Those are described in more detail below.

The best designs will apply all of these techniques. Our most ambition performance goal (t_{CLK} = 3 ns) will require careful attention to all aspects of your design. Good luck!

Carry-select adders

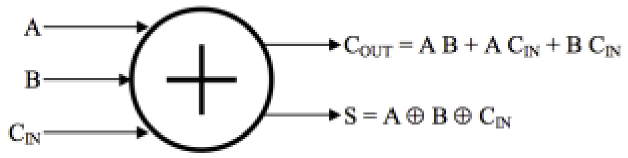
Idea: do two additions; one assuming the carry-in is 0, and the other assuming the carry-in is 1. Use a MUX to select the appropriate answers when the correct carry-in is known:

http://en.wikipedia.org/wiki/Carry-select_adder

With one stage (i.e., using carry-select on the top sixteen bits ofyour adder driven by the carry-out from the bottom 16-bits) the circuit is 50% larger but almost twice as fast as ripple-carry adder. With multiple, variable-sized blocks, t_{PD} approaches sqrt(N) where N is the number of bits in the operand. Carry-select can be combined with carry-lookahead (see below) for even greater performance improvements.

Carry-lookahead adders

The basic building block used in the ripple-carry adder is the **full adder**, the 3-input, 2-output combinational logic circuit shown below:

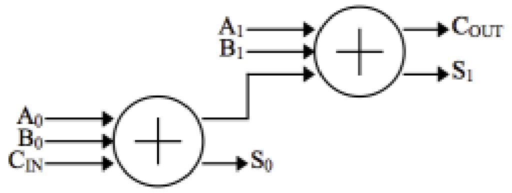

A
B
C_{IN}
$C_{OUT} = A\,B + A\,C_{IN} + B\,C_{IN}$
$S = A \oplus B \oplus C_{IN}$

Since the C_{OUT} signals of each full adder are in the critical path, let's see if we can improve the adder's performance by generating the C_{OUT} signals more quickly. First let's rewrite the equation for C_{OUT}:

$$C_{OUT} = A\,B + A\,C_{IN} + B\,C_{IN} = A\,B + (A + B)\,C_{IN} = G + P\,C_{IN}$$

where $G = A\,B$ is true if a carry is *generated* by the full adder and $P = A + B$ is true if the carry is *propagated* by the full adder from C_{IN} to C_{OUT}. Actually the propagate signal is sometimes defined as $P = A \oplus B$ which won't change the computation of C_{OUT} but will allow us to express S as a simple function of P and C_{IN}: $S = P \oplus C_{IN}$. Note that P and G depend only on A and B and *not* on C_{IN}.

We can generalize the notion of P and G to blocks of several bits. For example, consider a two-bit adder:



Define the block generate signal $G_{0,1}$ as $G_{0,1} = G_1 + G_0\,P_1$, i.e., the 2-bit block will generate a carry if a carry is generated in bit 1 (G_1) or if a carry is generated in bit 0 and propagated by bit 1 ($G_0\,P_1$). Similarly we can define the block propagate signal $P_{0,1}$ as $P_{0,1} = P_0\,P_1$, i.e. C_{IN} will be propagated to C_{OUT} only if both bits are propagating their carry-ins.

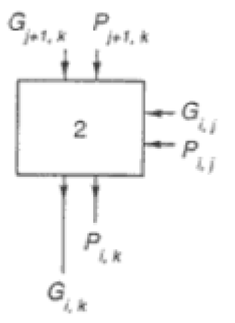
Building on this idea, we can choose the maximum fan-in we want for our logic gates and then build a hierarchical carry chain using these equations:

$$C_{J+1} = G_{IJ} + P_{IJ}C_I$$

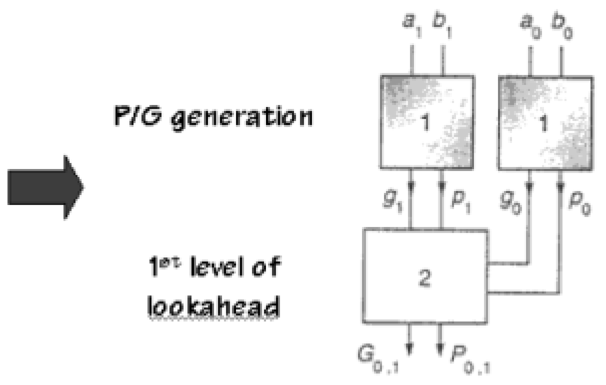
$$G_{IK} = G_{J+1,K} + P_{J+1,K}\,G_{IJ}$$

$$P_{IK} = P_{IJ}\,P_{J+1,K} \quad \textit{where } I < J \textit{ and } J+1 < K$$

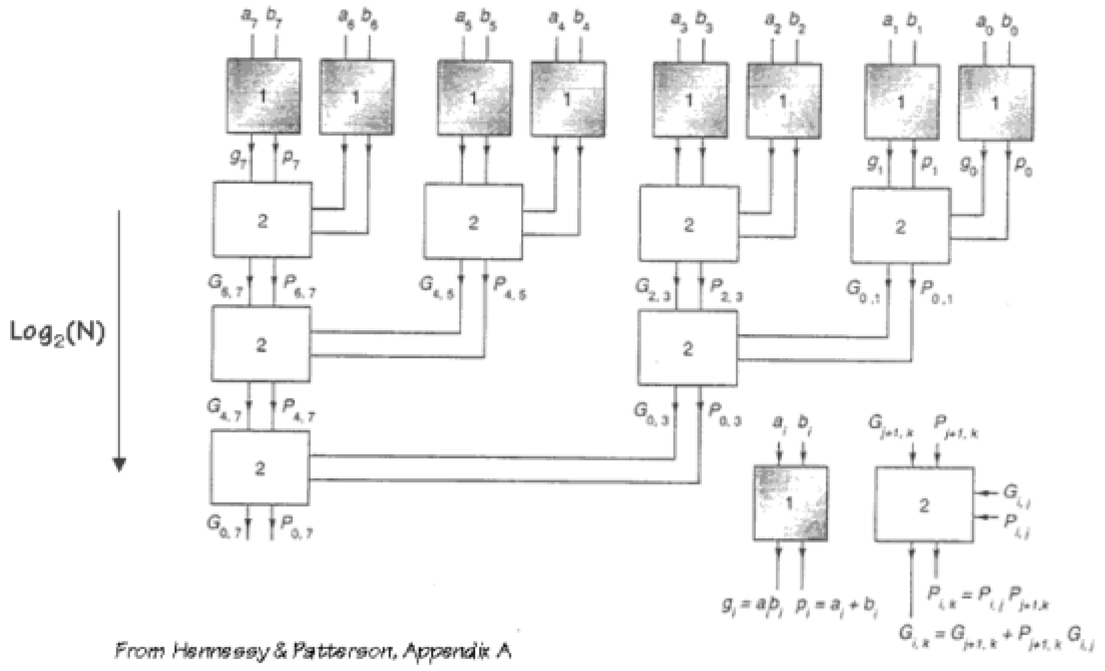
"generate a carry from bits I thru K if it is generated in the high-order (J+1,K) part of the block or if it is generated in the low-order (I,J) part of the block and then propagated thru the high part"



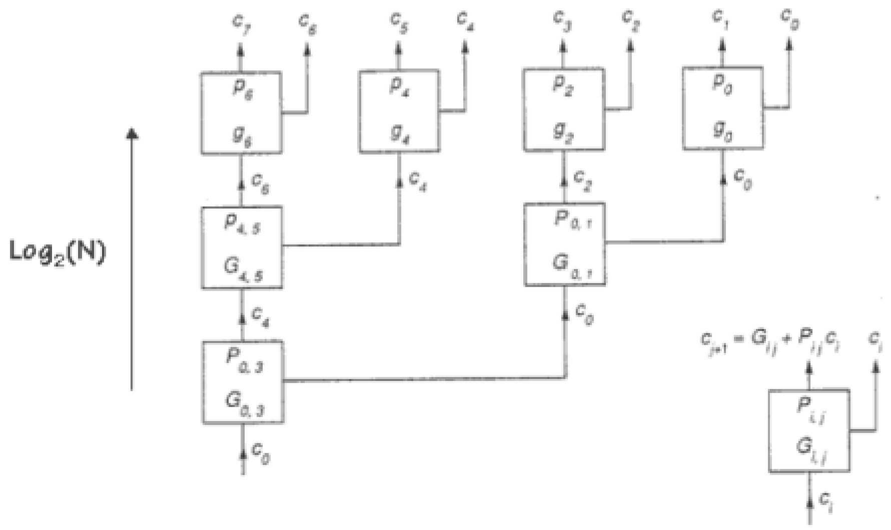
Hierarchical building block



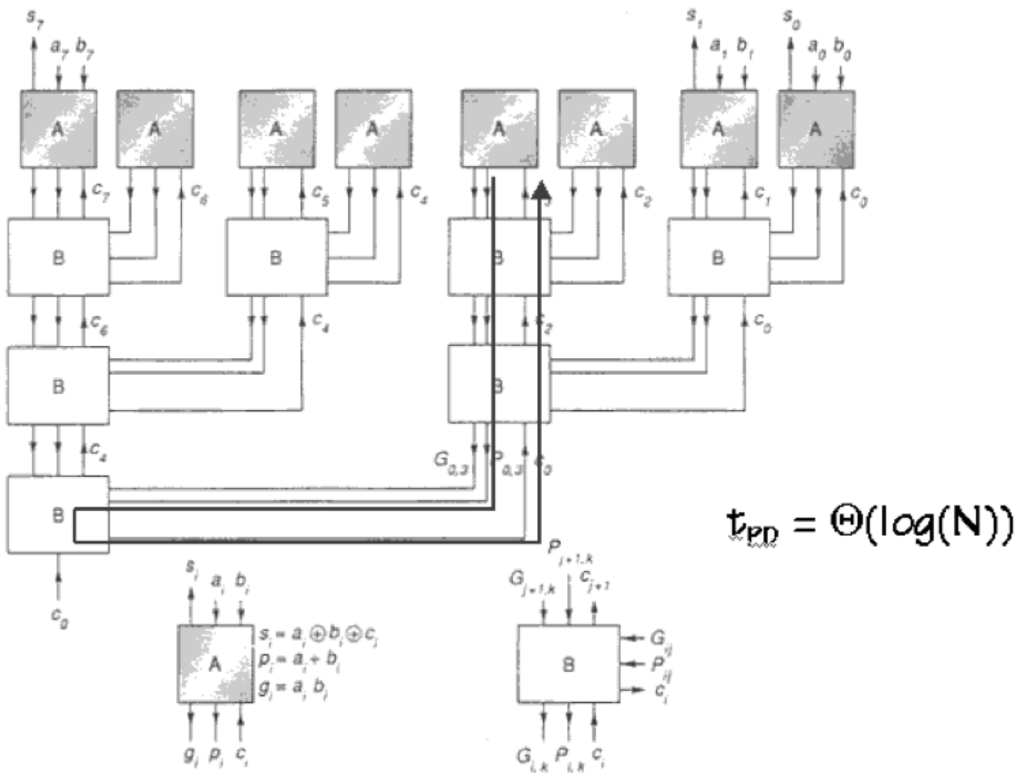
We can use this building block to construct P/G signals for any width operands. The following figure shows how this works for 8-bit operands using 2-input logic gates:



We can then use the P, G, and carry-in signals at each level to generate carry-in signals for previous levels of the hierarchy, as shown in the following diagram for an 8-bit adder:



We can combine P/G generation and carry generation into a single building block, leading the following diagram for a complete 8-bit carry-lookahead adder:



In order to use only inverting logic in the building blocks, it's common to have two versions: one that takes inverted inputs and produces non-inverted outputs, and another that takes non-inverted inputs and produces inverted outputs. These blocks can be used on alternative levels of the carry look ahead tree.

http://en.wikipedia.org/wiki/Carry-lookahead_adder

Part 2: Translating Gate Netlist to Conjunctive Normal Form (CNF)

Instructions:

1. Select one output bit that helps you proof the optimized circuit is equivalent to the original circuit.
2. From the gate netlist, derive the Boolean expression in conjunctive normal form (CNF).
3. You may write your own software or use and existing ones to do the translation from gate netlist to CNF.

Combinational Equivalence Checking

Combinational Equivalence Checking (CEC) is a fundamental technique to check digital circuits. The purpose of CEC is to verify that the two combinational circuit designs implement the same Boolean functions. One technique to do CEC is using SAT (Satisfiability). Given two outputs y_A and y_B of two circuits A and B, it can be verified that y_A and y_B are equivalent by showing that $y_A \oplus y_B$ is unsatisfiable.

In order to use SAT solver, it is common to convert the circuit netlist into Conjunctive Normal Form (CNF). When dealing with multi-level circuits, it is important to have an efficient translation of a circuit structure into CNF. The algorithm below describes the translation process from Boolean logic into CNF.

1. Write Boolean equations in terms of two input gates.
2. From these Boolean equations, write down the implications.
3. Replace implications by disjunctions.

For example, consider a circuit with three inputs a, b, c, and output o. The output is given by a Boolean equation

$$o = a + bc$$

In the first step, we write Boolean equation in terms of two input gates. In order to do this, we introduce a new variable t.

$$o = a + t$$

$$t = bc$$

In the second step, we translate these equations into their implications.

$o = a + t$	$o \rightarrow (a + t)$
	$a \rightarrow o$
	$t \rightarrow o$
$t = bc$	$t \rightarrow b$
	$t \rightarrow c$
	$bc \rightarrow t$

In the third step, the six implications are replaced by disjunctions to obtain CNF.

$$(\bar{o} + a + t)(\bar{a} + o)(\bar{t} + o)(\bar{t} + b)(\bar{t} + c)(\bar{b} + \bar{c} + t)$$

or in another form as

$$(\neg o \lor a \lor t) \land (\neg a \lor o) \land (\neg t \lor o) \land (\neg t \lor b) \land (\neg t \lor c) \land (\neg b \lor \neg c \lor t)$$

CNF Output Format

You only need to proof that the optimized circuit is equivalent with the original circuit for one output bit of the 32-bit adder. You have to choose which output bit you want check for equivalence, and you will need to explain your choice during the demo.

You need to submit the final CNF output text file in the format specified in the following link.

CNF format: https://app.box.com/s/opgbiv7310szl97nzv5devu3gi7qzbe3

Your software in Deliverable 2 must be able to take this format as input for your SAT solver.

Testing

You can test and debug your result using the following software: FindSolsSAT.

Example: OR Gate. An OR gate has two inputs and one output, and can be written as

$$o = a + b$$

The implications are:

$$o \to a + b$$
$$a \to o$$
$$b \to o$$

And so it can be written in CNF clauses as:

$$\bar{o} + a + b$$
$$\bar{a} + o$$
$$\bar{b} + o$$

Let a = 1, b = 2, and o = 3, the Input file will be written as

```
c test OR gate
p    cnf    3    3
-3   1      2    0
-1   3      0
-2   3      0
```

Save the input file as “testorgate.cnf”. Now, you can run the FindSolsSAT to get the solutions by typing:

```
java -jar findsolssat.jar testorgate.cnf
```

It will print out the output as:

```
Solution : 1
-1 -2 -3
Solution : 2
-1 2 3
Solution : 3
1 -2 3
Solution : 4
1 2 3
```

Note that negative number means that it is logic False, while positive number means that it is logic True. Looking at the solutions, we can find that it does display truth table for OR gate. To build the truth table from the solutions, put 0 if you see the literal is a negative number and 1 if it is a positive number.

a (1)	**b (2)**	**o (3)**
0	0	0
0	1	1
1	0	1
1	1	1

Using Existing Softwares to Create CNF

Instructions:

1. Read the documentation at: http://users.ics.aalto.fi/tjunttil/circuits/
2. Recreate the original 32-bit adder and the optimized circuit using the constrained Boolean circuit. Example is shown below. Save the file with ".bc" extension.

   For an example, we will create a simple OR gate BC file. Create a new text file, and save it as "or1.bc". Edit the file and key in the following:

   ```
   BC1.0
   Y := OR(A,B);
   ```

3. Go to http://10.1.3.26:8000/bc2cnf/
4. Click "Choose File" and select your BC file.
5. Enter your Student ID and click "Submit" to produce CNF.
6. Copy the text under "CNF Output" to a text file with an extension ".cnf".
7. Edit the CNF file to add the constraints above: $y_A \oplus y_B$

Instead of using the Web interface, you can also download the software to some Unix Machine (Mac OS X/Linux):

1. Download, compile, and install BC package from: http://users.ics.aalto.fi/tjunttil/circuits/BCpackage-0.35.zip
2. Run bc2cnf to produce the CNF as follows:

   ```
   $ bc2cnf –v –nosimplify –nocoi        or1.bc        or1.cnf
   ```

   The file or1.cnf will contain the CNF.