Table of Contents

List of Figures

List of Tables

1 Introduction

Multiplication is an important part of real-time digital signal processing (DSP) applications ranging from digital filtering to image processing. The multipliers used in such applications require many different operand size. An efficient way to design multipliers with different sizes is through the design of the parameterized cells.

A parameterized cell is a graphic, programmable cell that lets you create a customized instance each time you place it by allowing the designer to specify certain parameters. The parameter is a setting that can control the size, shape, or contents of the cell instance. The big advantage in using parameterized cell is that you can speed up the time of chip design by eliminating the need to create a lot of chips with the same function but different sizes. Also, parameterized cells can save disk space by creating a library of cells for similar parts that are all used the same cells.(for example in my design the regular multiplier, complex multiplier and multiplier-accumulator are almost use the same cells). However, parameterized design is not the only considerations; low power dissipation and small chip area are also needed because of the dense packing of transistors in today's DSP chips.

The main objective of this thesis is to design a parameterized cell library by using cadence CAD tool, call "parameterized cell (Pcell)", to automatically generate a 16x16-bit

multiplier, a 12x12-bit complex number multiplier and a 12x12-bit complex number multiplier-accumulator(MAC) for the second-generation digital backend receiver. The three main considerations for the design are a high multiplication speed, low power dissipation, and a small rectangular chip area.

All the designs are using SGS-THOMPSON MICROELECTRONICS's HCMOS7 0.25um technology. All layout and logic testing are done with Cadence CAD tools with Unicad design kit, and EPIC. All delay and power simulation are done with Meta Software's HSpice and PowerMill.

Chapter Two describes the design issues. First, it describes the architectures of the multiplier, complex multiplier, and complex multiplier accumulator. Second, it discusses the floor plan and layout. Third, it discusses the circuit design of the individual components (leaf cells). The last, it discusses the area, propagation delay and power dissipation. The last, it discusses the power reduction via the technology improvement.

Chapter Three describes the Pcell issues. First, it provides an overview of Pcell program. Second, it discusses the advantages and disadvantages of abutment. Third it presents a pcell design tutorial.

Chapter Four summarizes and concludes the report.

2 Multiplier Architecture and Leaf Cell Circuit Design

2.1 Multiplier

2.1.1 Architecture

The multiplier has two stages, the first stage consists of booth encoders which drive partial product generators which in turn drive a carry-save addition array to produce two final partial products. In the second stage, the two final products are added to form the final product through a ripple-carry adder. The multiplication architecture is shown in Figure 2-1.

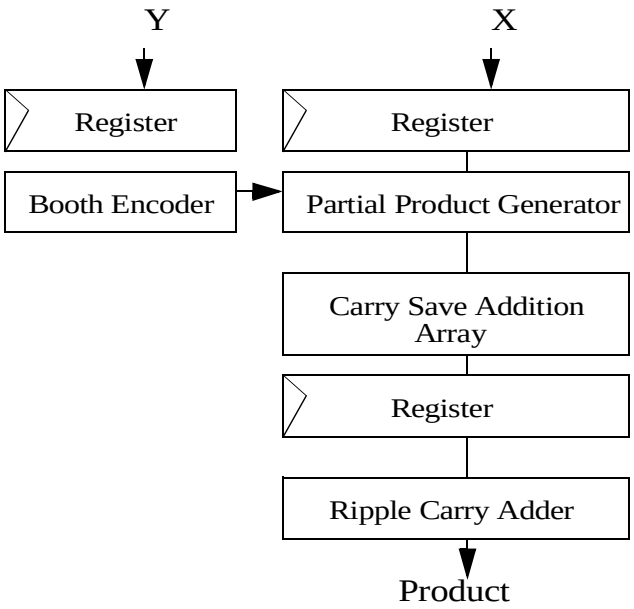


Figure 2-1: Architecture of Multiplier

The booth decoding algorithm and carry-save addition array were chosen because of two reasons which are further discussed in the next sections (2.1.1.1 and 2.1.1.2). First, the

booth algorithm is easy to handle the 2's complement number, and it requires half the number of partial products. Second, carry save addition array with modified booth algorithm results in a much more regular structure. These are more suitable for parameterized design.

2.1.1.1 Modified Booth Encoding Algorithm

The booth encoding algorithm is a bit-pair encoding algorithm that generates partial products which are multiples of the multiplicand. The booth algorithm shifts and/or complements the multiplicand (X operand) based on the bit patterns of the multiplier (Y operand). Essentially, three multiplier bits [Y(i+1), Y(i), and Y(i-1)] are encoded into nine bits that are used to select multiples of the multiplicand{-2X, -X, 0, +X, +2X}. The three multiplier bits consist of a new bit pair [Y(i+1), Y(i)] and the leftmost bit from the previously encoded bit pair [Y(i-1)]. Refer to table 2-1 for the details of the booth algorithm.

Table 2-1. partial product selection

Y(i+1)	Y(i)	Y(i-1)	Encoded digit	Operation Multiplicand(X)
0	0	0	0	0 * X
0	0	1	+1	+1 * X
0	1	0	+1	+1 * X
0	1	1	+2	+2 * X
1	0	0	-2	-2 * X
1	0	1	-1	-1 * X
1	1	0	-1	-1 * X
1	1	1	0	0 * X

For an mxn-bit multiplication, the booth algorithm produces n/2 [(n+1)/2, if n is odd] partial products, each has a length of (m+1) bits. This can half the number of partial products. It reduce the number of adders by 50% which results in a higher speed, a lower power dissipation, and a smaller area than a conventional multiplication array.

The general booth algorithm often uses sign extension which means that each partial product has its sign bit extended(repeated) to the leftmost MSB of the last partial product. The disadvantage of sign extension is that it increases the number of bits to add together. This require extra adders which decrease speed, increase power dissipation and increase area.

A modified version of the booth algorithm uses sign generation to eliminate sign extension. The sign extension is implements as follows:
1) Complement the sign or (m+1)th bit of each partial product.
2) Add 1 to the left of the sign bit of each partial product.
3) Add 1 to the sign bit of the first partial product.
The advantage of sign extension is that it doesn't create an extra bit vector or partial product. Because we can insert the ones into the cells with half adder. Therefore, no extra adders are needed to implement the sign generation scheme.

The modified booth encoding algorithm results in (n/2) [(n+1)/2, if n is odd] adder rows with each row consisting of m adders for a mXn-bit multiplication. This results in an extremely regular and rectangular multiplication architecture.

2.1.1.2 Carry-Save Addition Array

An example of an 8x8 carry-save addition array is shown in Figure 2-2. The basic idea behind the array is as follows:
1) Use only half adders in the first row (no partial product reduction).
2) Reduce the partial product from eight to seven with the second row.
3) Reduce the partial products from seven to six with the third row.
4)Continue this reduction process until there are only two final partial products.
Each reduction step (except the first non-reduction step) is perform by reducing the top three partial products to two partial products with an adder row. The rest of the partial products are left alone until the next reduction step.

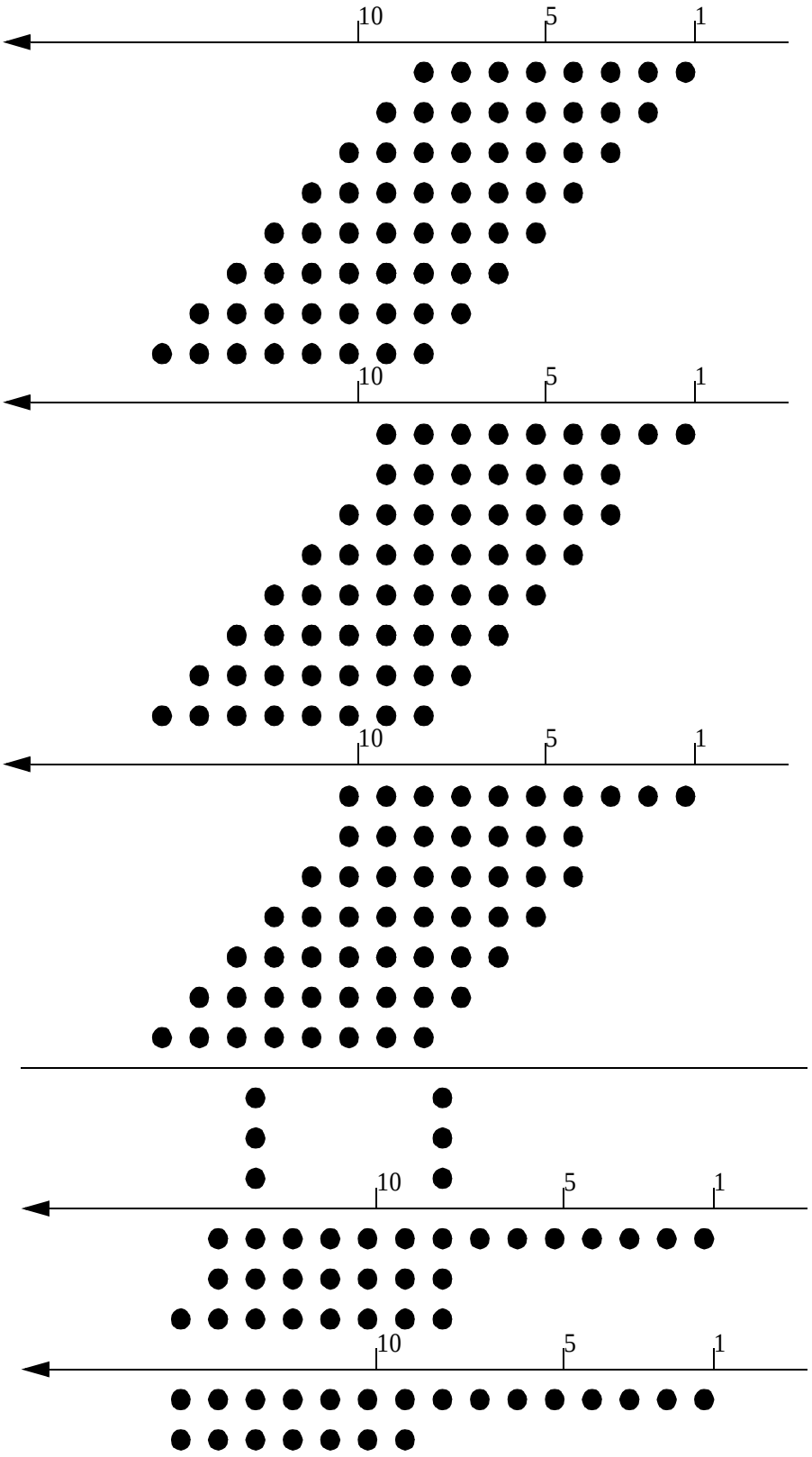


Figure 2-2: 8X8 Carry-Save Addition Array

Carry-save addition array has the advantage of being very regular (see figure 2-3). However, it is not very fast since it requires the maximum number of adder row (n-1) to generate the final two partial products for n initial partial products. Since the booth algorithm was chosen for the 12x12-bit multiplier, this means 6 initial partial products require 5 adder rows.

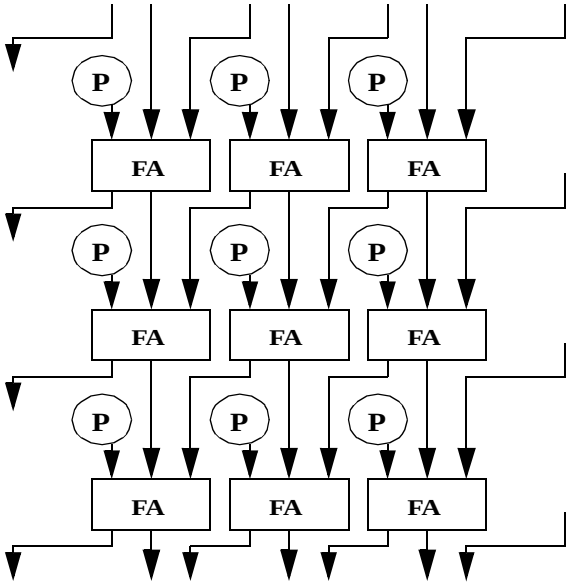


Figure 2-3: Carry Save Core Array

2.1.1.3 Rectangular Versus Parallelogram Arrays

It is clear that a rectangular multiplier is always smaller than a parallelogram one. However, is it still the case when a rectangular multiplier is imbedded in a datapath with other cells. This question exists due to the fact that the rectangular multiplier has LSB outputs exiting the array from the right while the parallelogram has all outputs exiting from the bottom.

In our receiver project, wordlength truncation will be done after multiplication. Assume t he pitch of all cells are identical (before the multiplier, in the multiplier, and after the multiplier), then a parallelogram array would always consume more area than a rectangular

array. The wasted area of the parallelogram array shown in Figure 2-4. For the rectangular multiplier, a suitable block router will have to be used to route output signals from the right of the array multiplier to the bottom.

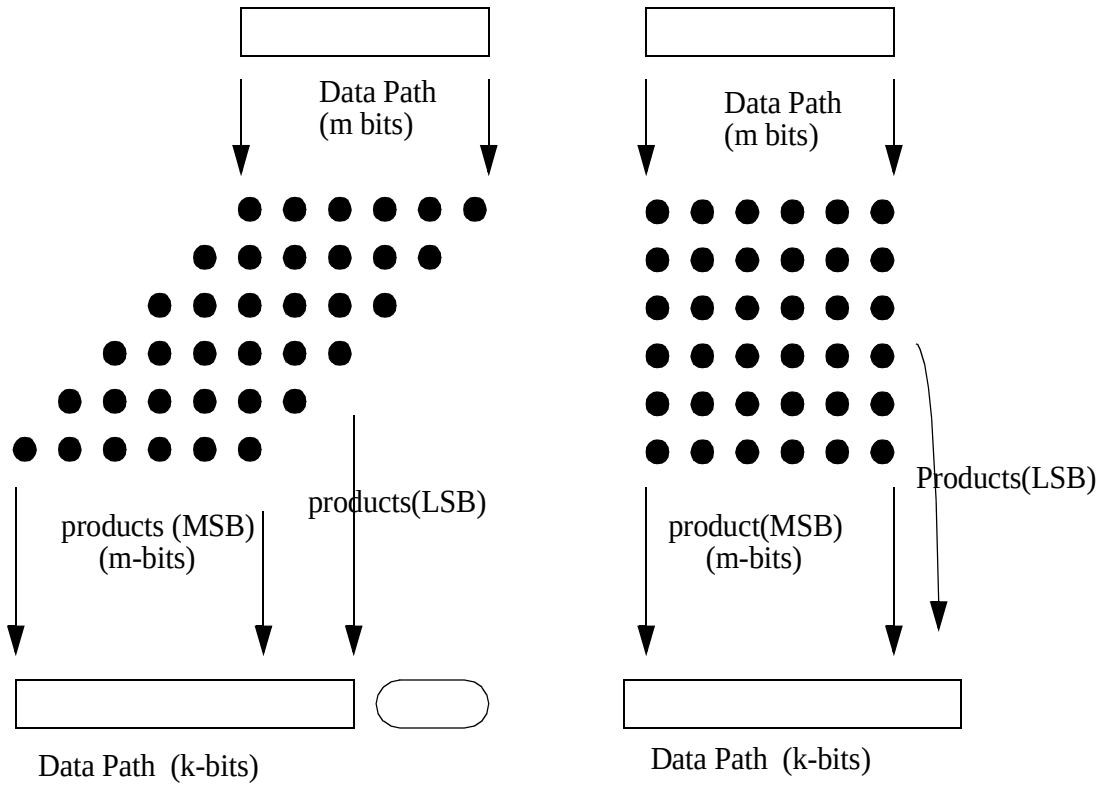


Figure 2-4: Parallelogram Versus Rectangular Arrays: Area Usage

Another problem with the parallelogram array is that it causes a shift in the datapath. The MSB bit into the multiplier will not be the same position as the MSB bit out of the multiplier. This doesn't happen in the rectangular multiplier. It is usually desirable in datapath to align all MSB bits.

Because of the arguments presented above, all the parameterized cells developed in this paper have been designed to generate rectangular multipliers and MAC.

2.1.2 Circuit Design

A parameterized cell for generating a carry save multiplier has been designed. Figure 2-5 shows the schematic and the layout of the 16-bit by 16-bit multiplier.

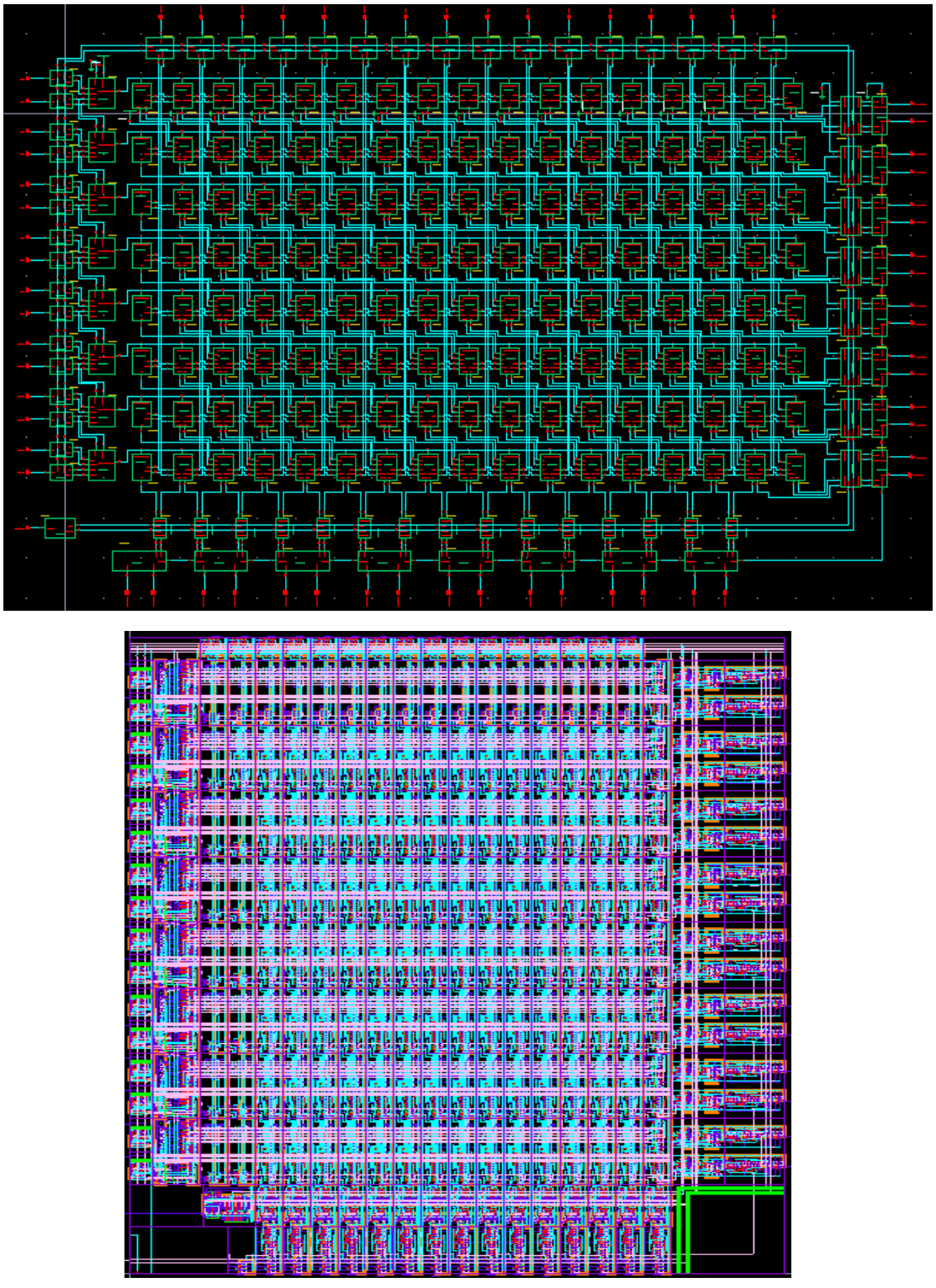


Figure 2-5: Schematic and the Layout of the 16x16b multiplier

2.1.2.1 Cell Description

The following are the contents of the cells that are used in the carry save multiplier. Note that routing is included in each cell so that by tiling these cell together, one get the entire multiplier (see Figure 2-6).

<table>
<tr><td colspan="2">spmpy5-0</td><td>reg1x</td><td>reg1x</td><td>reg1x</td><td>reg1x</td><td>reg1x</td><td>reg1x</td><td colspan="3">spmpy1-0</td></tr>
<tr><td>reg1y</td><td rowspan="2">btd-0</td><td rowspan="2">arya</td><td rowspan="2">aryc</td><td rowspan="2">aryc</td><td rowspan="2">aryc</td><td rowspan="2">aryc</td><td rowspan="2">aryc</td><td rowspan="2">aryf</td><td rowspan="2">reg3y-0</td><td rowspan="2">addery-0</td></tr>
<tr><td>reg1y</td></tr>
<tr><td>reg1y</td><td rowspan="2">btd-0</td><td rowspan="2">aryb</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryg</td><td rowspan="2">reg3y-0</td><td rowspan="2">addery-0</td></tr>
<tr><td>reg1y</td></tr>
<tr><td>reg1y</td><td rowspan="2">btd-0</td><td rowspan="2">aryb</td><td rowspan="2">arye</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryg</td><td rowspan="2">reg3y-0</td><td rowspan="2">addery-0</td></tr>
<tr><td>reg1y</td></tr>
<tr><td>reg1y</td><td rowspan="2">btd-0</td><td rowspan="2">aryb</td><td rowspan="2">arye</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryd</td><td rowspan="2">aryg</td><td rowspan="2">reg3y-0</td><td rowspan="2">addery-0</td></tr>
<tr><td>reg1y</td></tr>
<tr><td colspan="2">spmpy4-0</td><td colspan="2">ckd3</td><td>reg3x-0</td><td>reg3x-0</td><td>reg3x-0</td><td>reg3x-0</td><td>reg3x-0</td><td colspan="3" rowspan="2">spmpy2-0</td></tr>
<tr><td colspan="3">spmpy3-0</td><td colspan="2">adderxm-0</td><td colspan="2">adderx-0</td><td colspan="2">adderx-0</td></tr>
</table>

Figure 2-6: Cells of the 8x6 Multiplier

adderx-0, addery-0 --> one 2-bit adder.

adderxm-0 --> one full adder and two inverter.

arya, aryb, arye --> one partial product generator and one inverter.

aryc --> one partial product generator.

aryd --> one partial product generator and one full adder.

aryf, aryg --> one partial product generator, one full adder, and one 2- input nand gate.

btd-0 -->one booth encoder.

reg1x, reg1y, reg3x-0 --> one register.

reg3y-0 --> two registers.

ckd3 --> one clock line buffer.

spmpy[1-5]-0 --> wiring cells.

2.2.2.2 Circuits

The following are the circuits used in the carry save multiplier.

Full Adder

The full adder is used in the rows of the carry save addition array to add the partial products. It is the most important multiplier core leaf cell for two reasons. First, a large percentage of the core propagation delay is due to full adders. Second, a large percentage of the core leaf cells are full adders. Therefore, the full adder chosen must have a high speed, low power dissipation, and small area. Also, the full adder must have equal carry and sum propagation delays since the propagation delay of a carry save addition depends on both signals. If they are different, glitch will happen and waste the power.

A transmission gate full adder was used for three reasons. First, it has equal carry and sum propagation delays. Second, it has a higher speed than either a static CMOS full adder or a carry-select full adder. Third, it has a small area (few transistors).

A transmission gate full adder uses only transmission gates and static inverters. Its main component is a multiplexer with A as the select signal. The operation of a multiplexer circuit is described by the following logic equations:

Equation 2-1.
$$Y = \begin{pmatrix} B \text{ , if } & A = 0 \\ \bar{B}, \text{ if } & A = 1 \end{pmatrix}$$

These equations are implemented by the transistor schematic shown in Figure 2-7. The multiplexer essentially acts as a XOR circuit. If the value of A is 0, Y=1 when B=1, If A is 1, Y=1 when B=0.

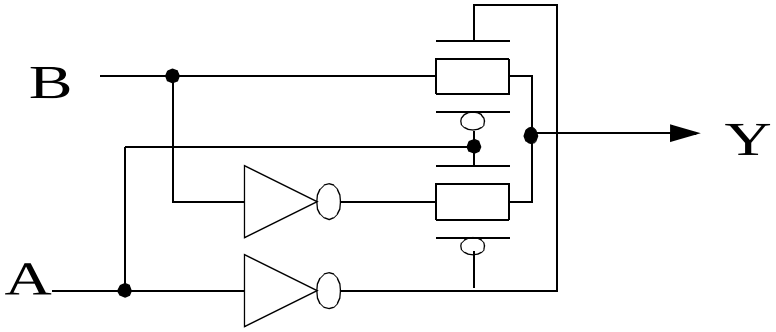


Figure 2-7: Transmission Gate Multiplexer

The operation of the full adder (which uses multiplexer to implement XOR) is described by the following logic equations:

Equation 2-2. $$Cout = \begin{pmatrix} \text{A or B, if A } \oplus B = 0 \text{ (A, B equal)} \\ \text{Cin, if A} \oplus B = 1 \text{ (A, B unequal)} \end{pmatrix}$$

Equation 2-3. $$Sout = \begin{pmatrix} \text{Cin, if A} \oplus B = 0 \text{ (A, B equal)} \\ \overline{Cin}\text{, if A} \oplus B = 1 \text{ (A, B unequal)} \end{pmatrix}$$

This equations are implemented by the logic schematic shown in Figure 2-8. The output of the XOR gate acts as the select signal for the transmission gate multiplexer on the output. Depending on the selection signal, the carry output (Cout) is either A or Cin the sum output (Sout) is either Cin or its complement.

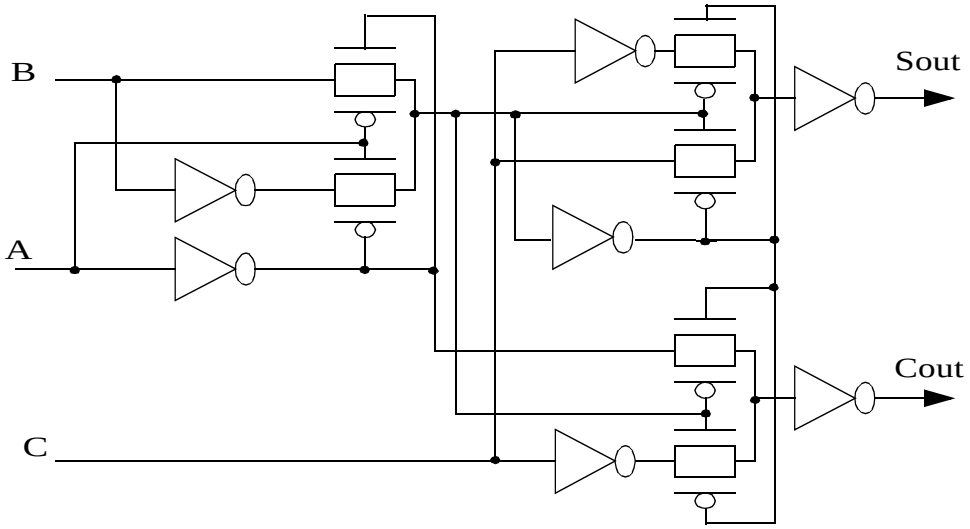


Figure 2-8: Schematic of Transmission Gate Full Adder

Booth Encoder

The booth encoder is used in the 3 to 9-bit modified booth encoding algorithm to encode the multiplier (Y operand). It requires three multiplier bits [Y(i+1), Y(i), Y(i-1)] as its input and generates nine select signals as its output to choose [+2X, +X, 0, -X, -2X]. These select signals drive the partial product generators in order to determine the operation on the multiplicand (X operand). The booth encoder must be buffered in order to drive multiple partial product generators which are interconnected by long routing bus. The logic schematic shown in Figure 2-9.

.

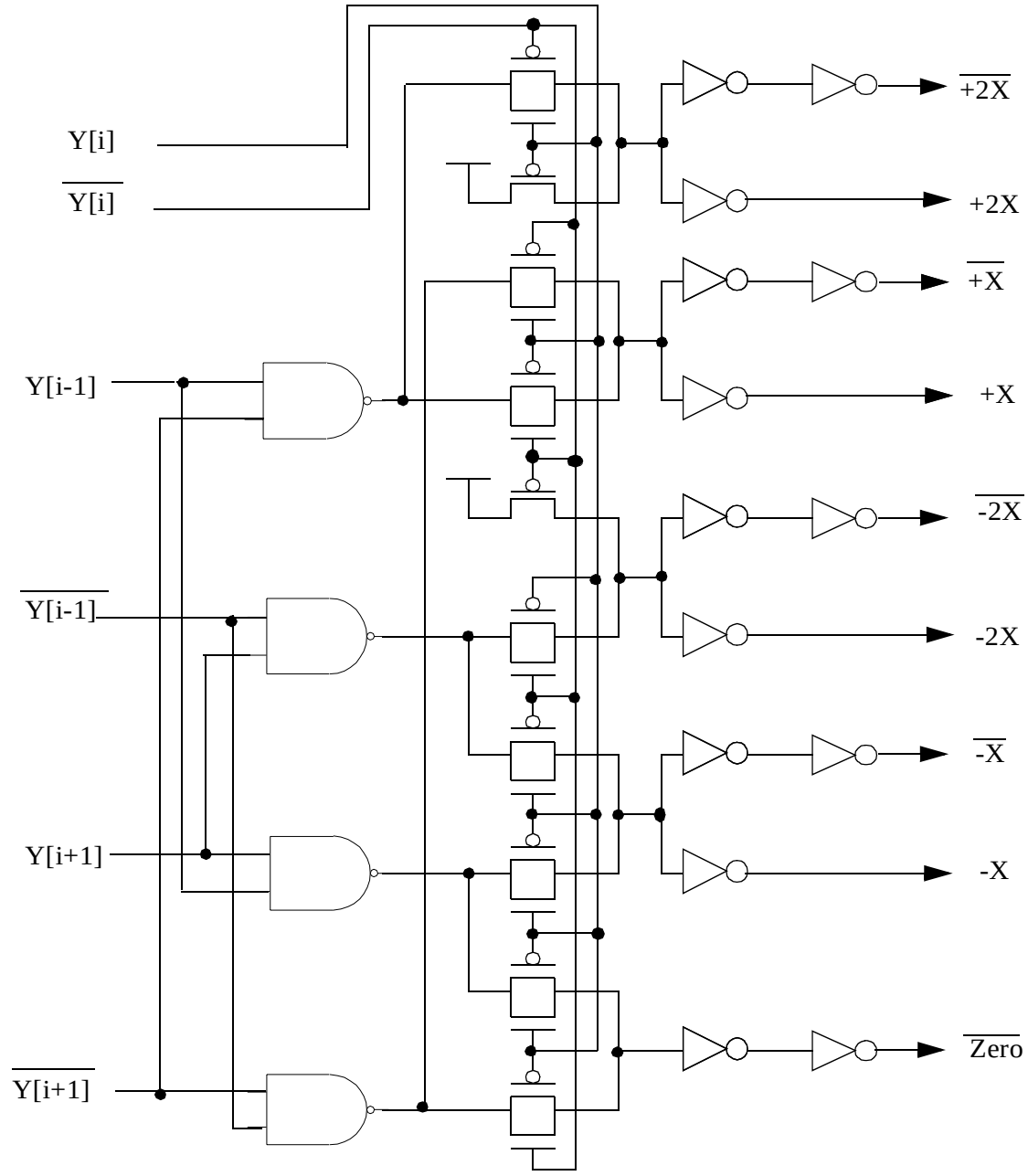


Figure 2-9: Schematic of Booth Encoder

3 to 9-bit modified booth encoder has more transistor number but all the select signals have the same propagation delay and it simplifies the logic and reduces the transistor number of partial product generator. Also, because all the propagation delays are the same, it can reduce the power which dissipated by the glitch. Another advantage of using

3 to 9-bit modified booth encoder is that each select signals only has one transistor load in each partial product generator, when you increase the wordlength of your multiplier, it will not degrade the performance of the multiplier too much.

Partial Product Generators

The partial product generator only uses eleven transistors. It actually is a multiplexer which controlled by the booth encoder selection signals. The output of the partial product generators are multiples of the multiplicand (-2X, -X, 0, +X, +2X). The logic schematic shown in Figure 2-10.

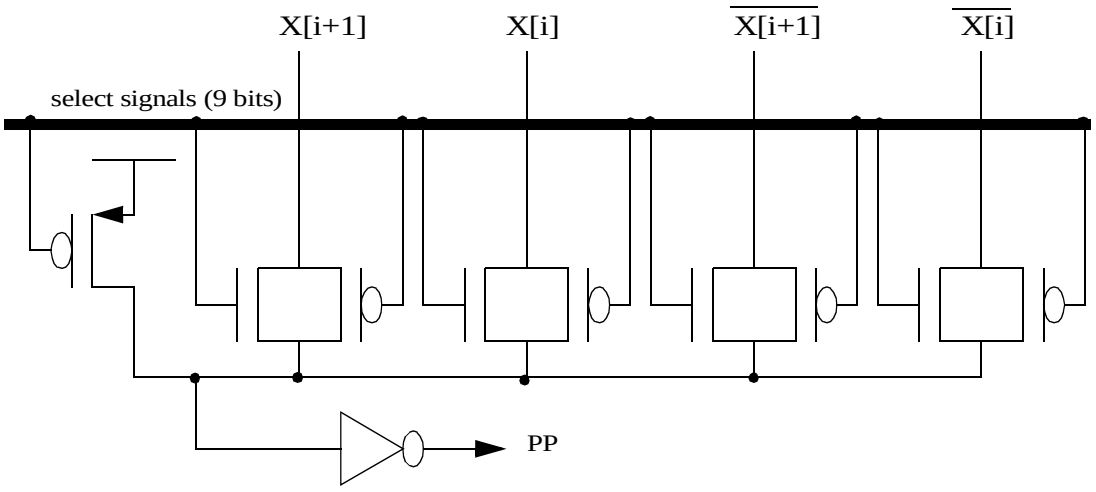


Figure 2-10: Partial Product Generator

Ones Generators

The ones generator is used in the modified booth encoding algorithm to help generate the two's complement of the multiplicand (X operand). When the operation is -X or -2X, we need to add one to the LSB.

The ones generator is controlled by two of the booth encoder selection signals(-2X and -X). The operation of the ones generator is described by the following logic equation.

Equation 2-4. $ONE = (\overline{-2X})$ NAND $(\overline{-X})$

The transistor schematic of ones generator is shown in Figure 2-11.

.

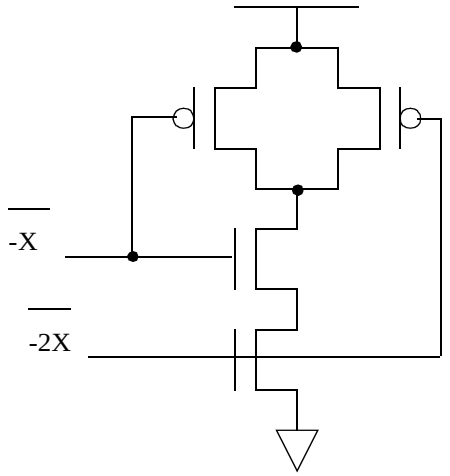


Figure 2-11: Ones Generator (NAND gate)

Ripple Carry Adder

The general structure of an n-bit ripple carry adder is shown in Figure 2-12. The A and B input vectors represent the final two n-bit partial products generated by the multiplier addition array. The carry out (Cout) propagation delay is due to the carry signal rippling serially through the block (initiated by the Cin input). Therefore, a full adder with a fast serial-carry delay is needed in the carry block.

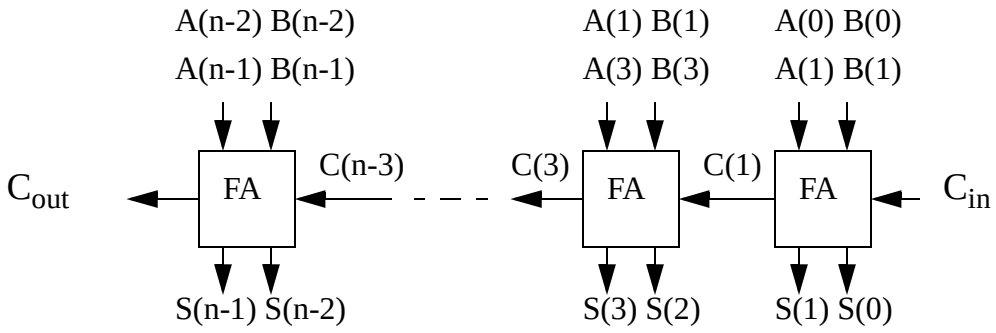


Figure 2-12: n-Bit Ripple Carry Block (each block is a two bit adder)

The logic schematic of ripple carry block shown in Figure 2-13. The carry signal propagates only through one transmission gate for each bit. However, due to the distributed RC effect and low supply voltage (less than 1 volt), an inverter is needed to reduce the degradation of the carry signal. Thus, each ripple carry block is a two-bit adder, the second adder is a complementary carry path adder in order to save one inverter.

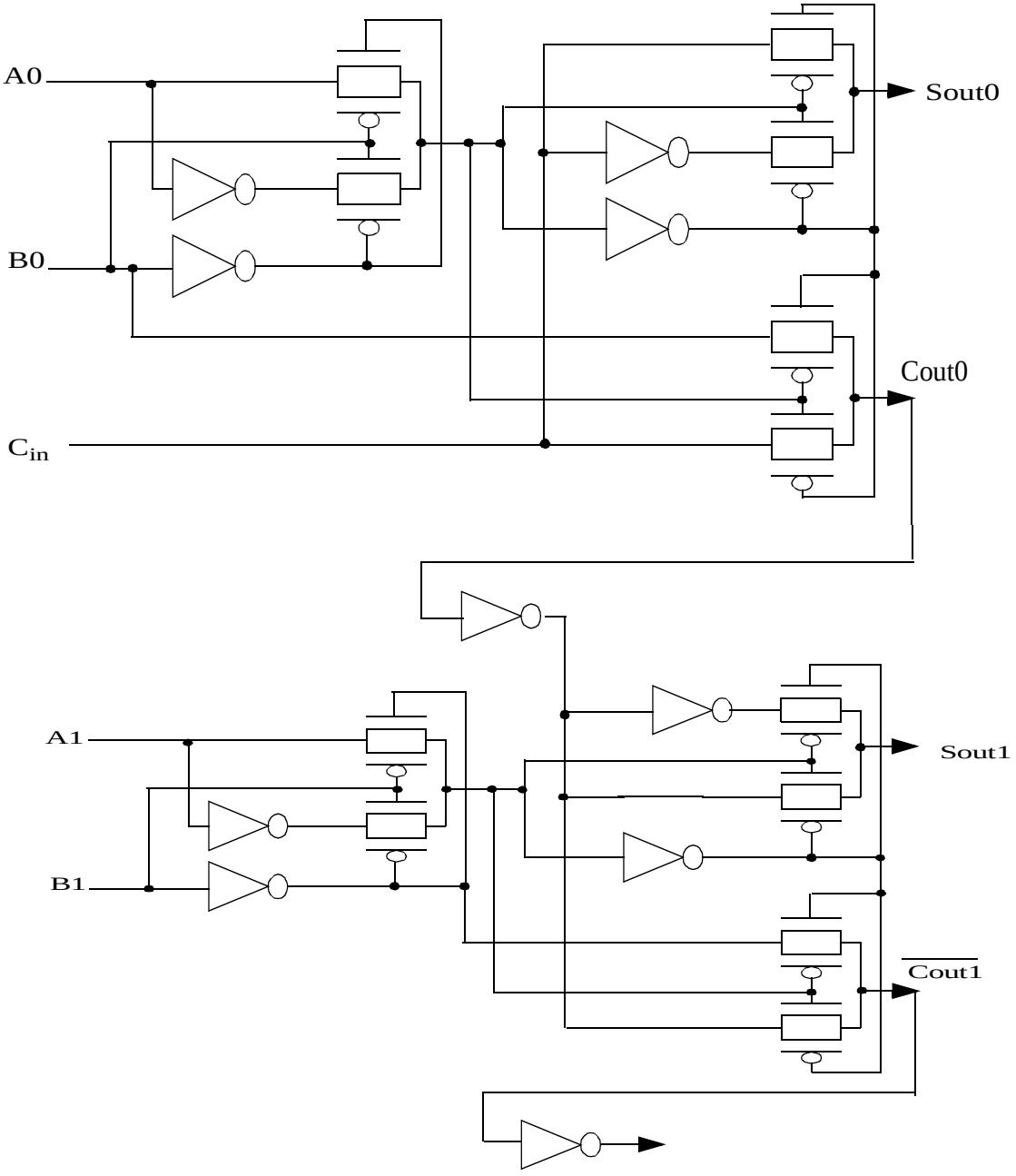


Figure 2-13: Schematic of Ripple Carry Block

2.2 Complex Multiplier

2.2.1 Architecture

The complex multiplication is based on the formula

$$(A + Bj)(C + Dj) = (AC - BD) + (\mathrm{AD+BC})\mathrm{j}$$

The complex multiplier has two stages, the first stage consist of four carry-save addition arrays which generate two final partial products for AC, BD, AD, and BC, and 4:2 compressor which reduce the four partial products to two. In the second stage, it has two sets of ripple carry adder to form the final product. The complex multiplier architecture is shown in Figure 2-14.

Most of the hardware and delay savings from merging the partial products of the 4 carry-save addition array is from removing the ripple carry adder at the second stage of the individual multipliers. We can get the 2 final partial products of AD+BC and AC-BD from the 4:2 compressor, without calculating the product of AD, AC, and BC, and BD. Another advantage is that the layout is more regular, since each bit needs a 4:2 compressor. Also with the addition of observation points at the inputs of 4:2 compressors, each multiplier can be tested individually.

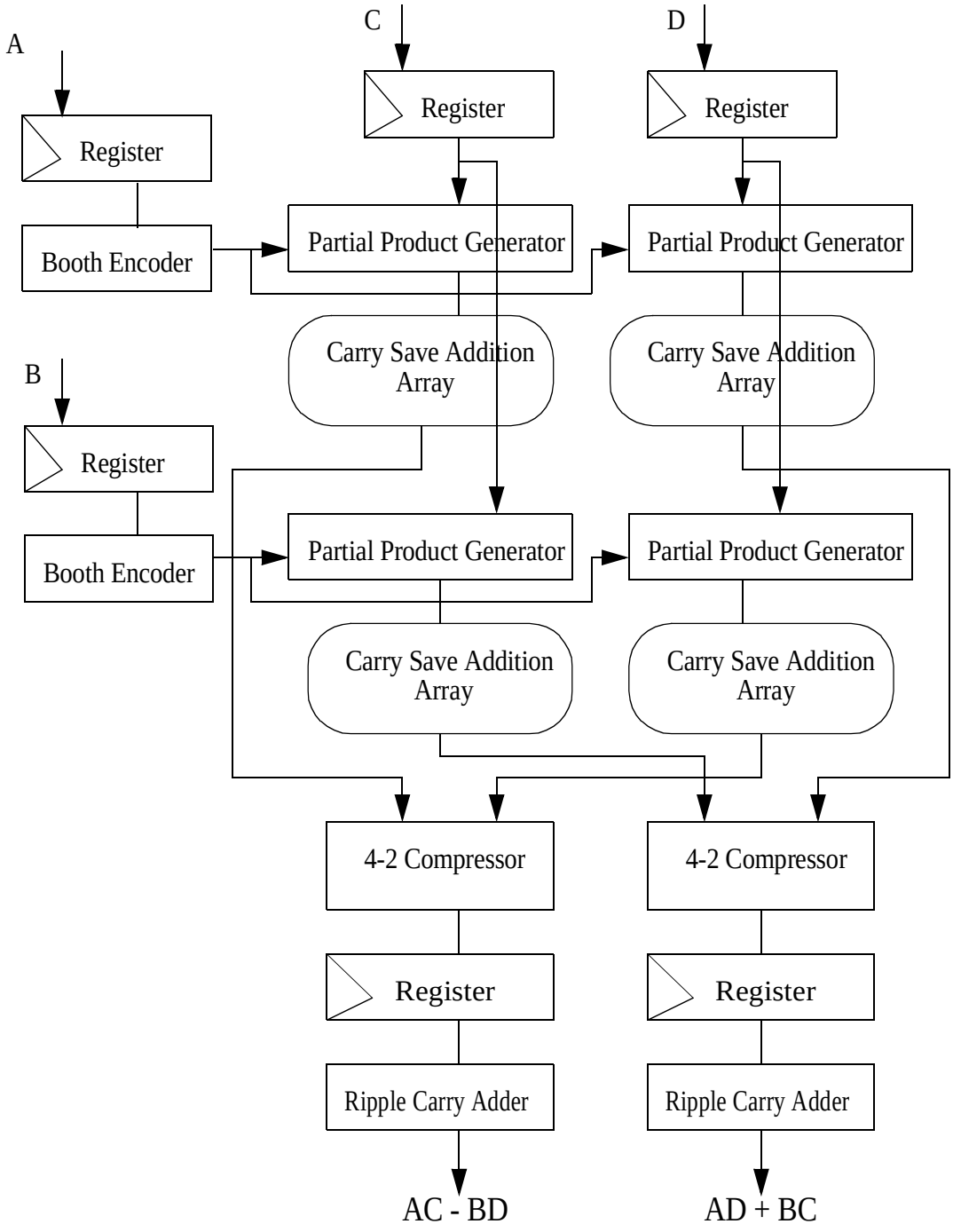


Figure 2-14: Architecture of Complex Multiplier

2.2.2 Circuits Design

A parameterized cell for generating a carry save multiplier has been designed. Figure 2-15 shows the schematic and the layout of the 12-bit by 12-bit complex multiplier.

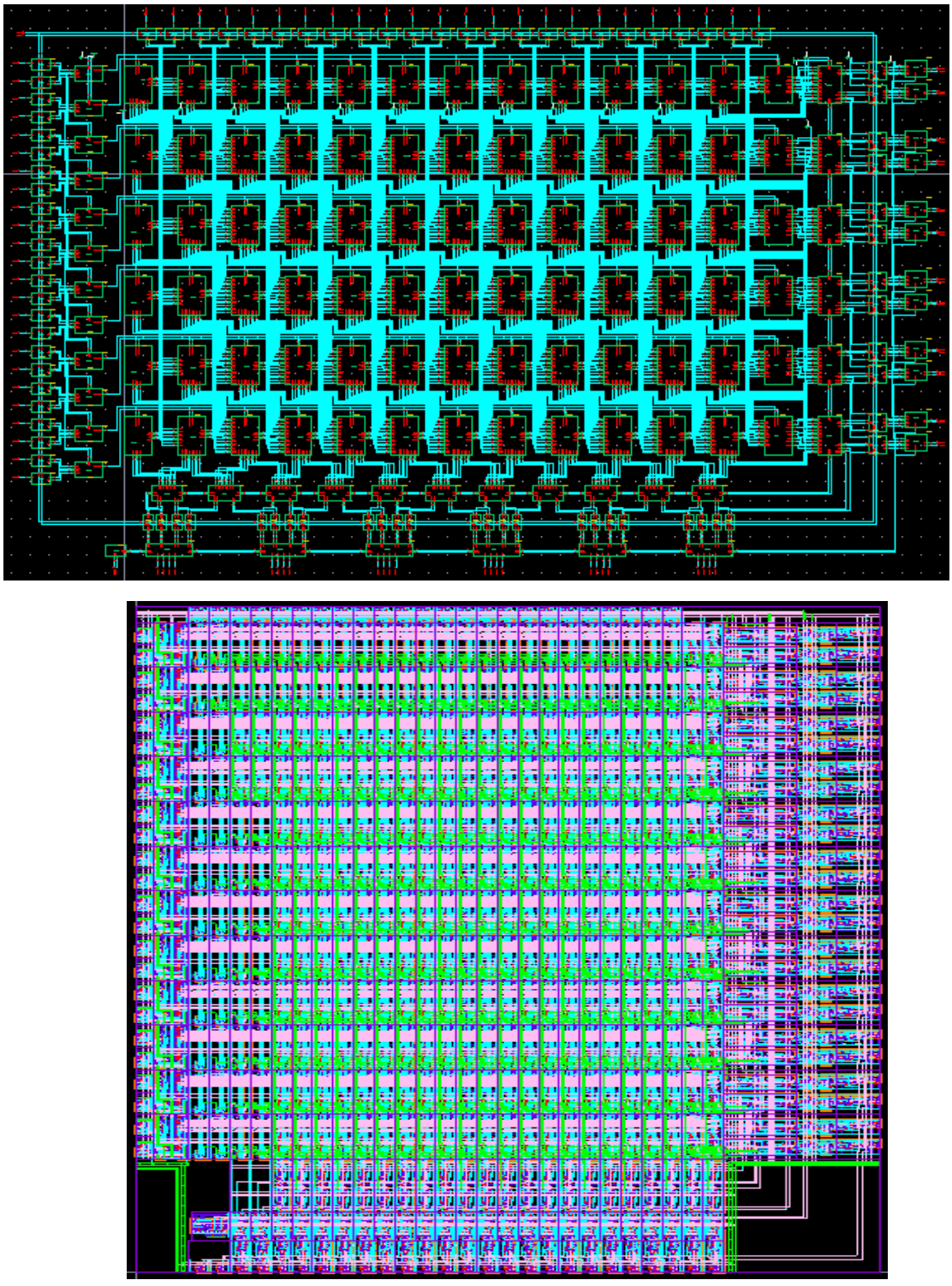


Figure 2-15: Schematic and Layout of the 12x 12 Complex Multiplier

2.2.2.1 Cell Description

The following are the contents of the cells that are used in the complex multiplier.Same as the integer multiplier, those cells at the four corners are always necessary. Because the routing is included in each cell so that by tiling these cells together, one gets the entire complex multiplier (see figure 2-16).

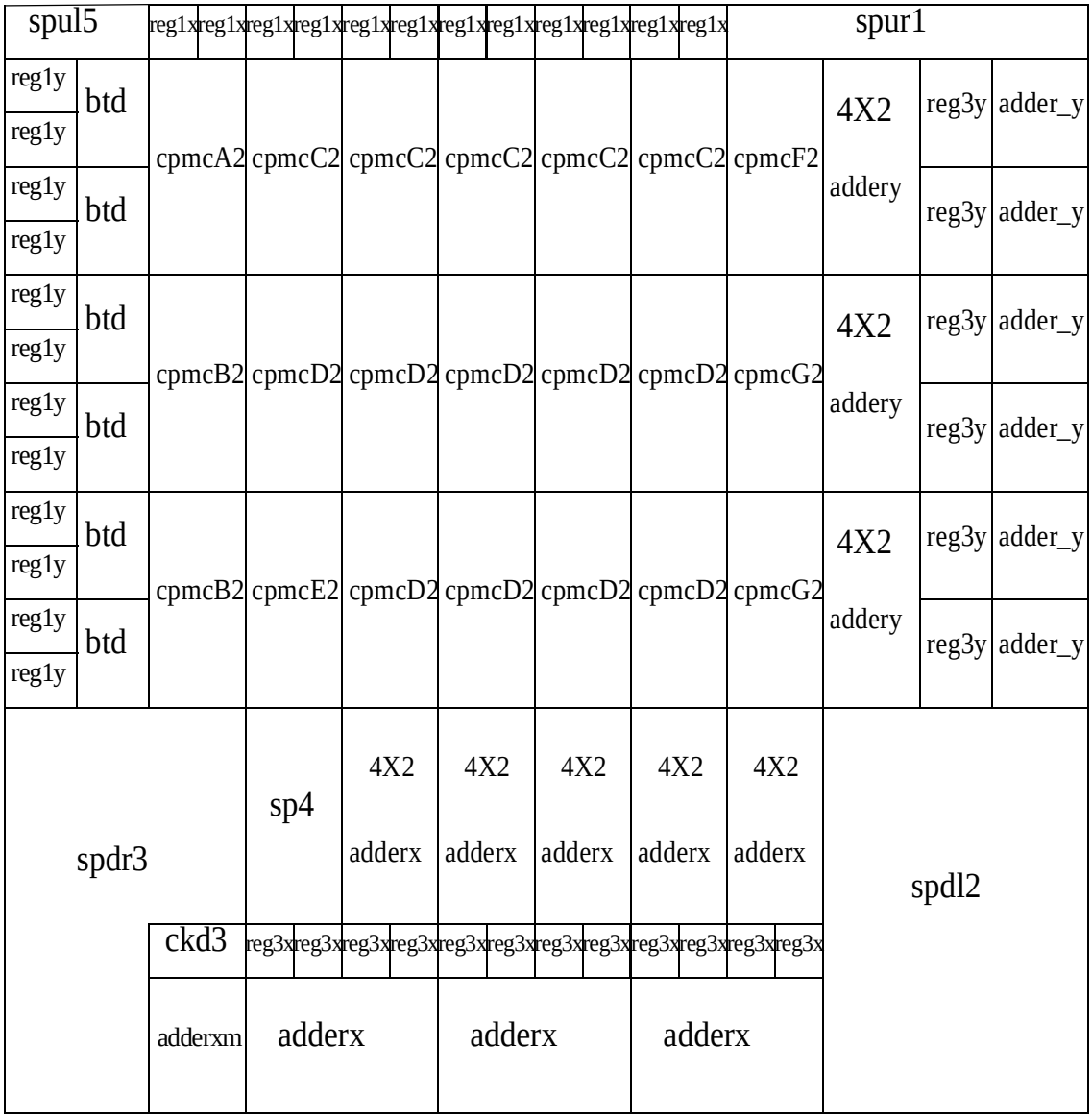


Figure 2-16: Cells of the 6X6 Complex Multiplier

addery --> two adders.

adder_x --> four adders.

4x2adderx --> two 4:2 compressors.

4x2addery --> four 4:2 compressors.

adderxm --> four inverters.

cpmc[A-G]2 --> four ary[a-g]s.

btd --> one booth encoder.

reg1x, reg1y, reg3x --> one register.

reg3y --> two registers.

ckd3 -- > clock line buffer.

spur1, spdl2, spdr3, spul5 --> wiring cells.

2.2.2.2 Circuit

4:2 Compressor

Figure 2-17 is a block diagram of the 4:2 compressor. The truth table for the 4:2 compressor is shown in Table 2-2. 4:2 compressor actually has five inputs and three outputs. It is different from a 5:3 counter which takes in five inputs of the same weight and produces three outputs of different weights. The sum output of the 4:2 has weight 1 while the carry and Cout both have the same weight of 2. In addition, the Cout output must not be a function of the Cin input, otherwise a ripple carry could occur.

Table 2-2. Truth Table For The 4:2 Compressor

n is number of inputs (from In1, In2, In3, In4) which =1, Cin is the input carry from the adjacent bit slice, Cout and carry both have weight 2, and sum has weight 1.

n	Cin	Cout	Carry	Sum
0	0	0	0	0
1	0	0	0	1
2	0	*	*	0
3	0	1	0	1
4	0	1	1	0
0	1	0	0	1
1	1	0	1	0
2	1	*	*	1
3	1	1	1	0
4	1	1	1	1

*Either Cout or Carry may be ONE for two or three inputs equal to 1 but not both.

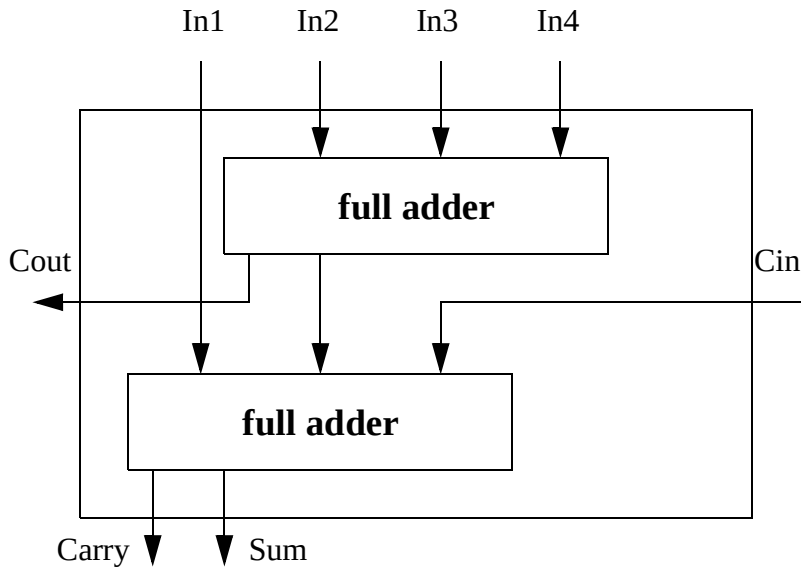


Figure 2-17: 4:2 Compressor implemented with two full adders

In order to deal with subtraction (AC-BD), we need to inverse the two final partial products of BD and add two ones to the LSB. Instead of adding two inverters, we modified the 4:2

compressor (see figure 2-18). The advantage is that we save numbers of inverters (2*(m+n-1)), also both real part number and imaginary part number have the same propagation delay. The disadvantage is that makes the circuits irregular.

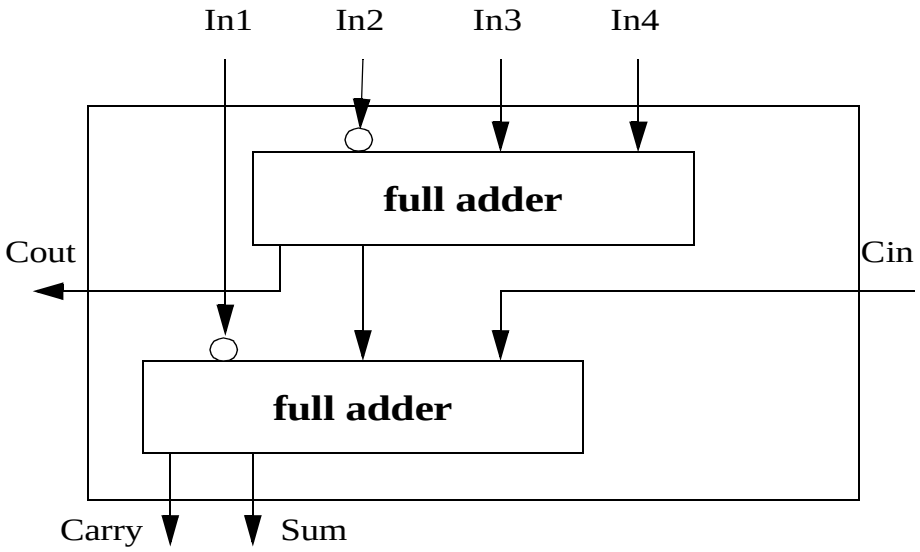
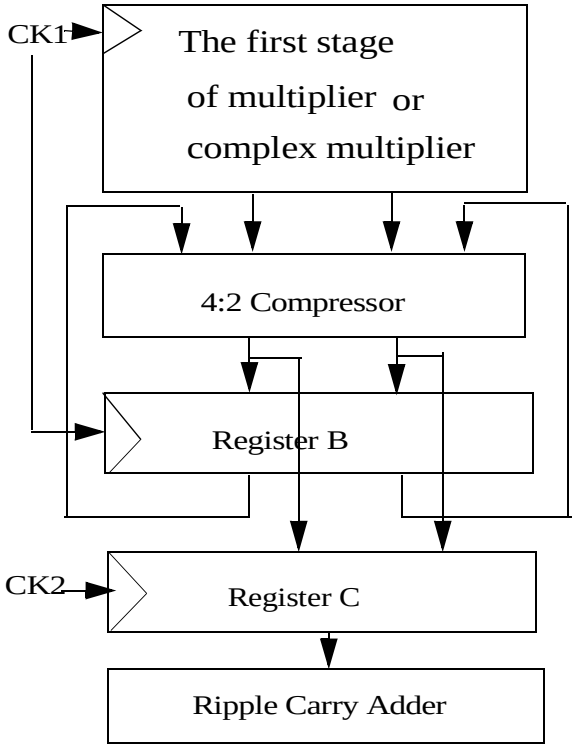


Figure 2-18: Modified4:2 compressor

2.3 Multiplier Accumulator

2.3.1 Architecture

To make MAC (Multiplier Accumulator) and multiplier common, both integer and complex number MAC use the same architecture and same hardware of their multiplier. The accumulation is implement by using a row of 4:2 compressors with reset and another row of registers to store the temporary value. The MAC architecture is shown in Figure 2-19.



Circuit Figure 2-19: Architecture of Multiplier Accumulator

2.3.2 Circuit Design

Parameterized cell for generating a carry save multiplier has been designed. Figure 2-20 shows the schematic and the layout of the 12-bit by 12-bit complex MAC.

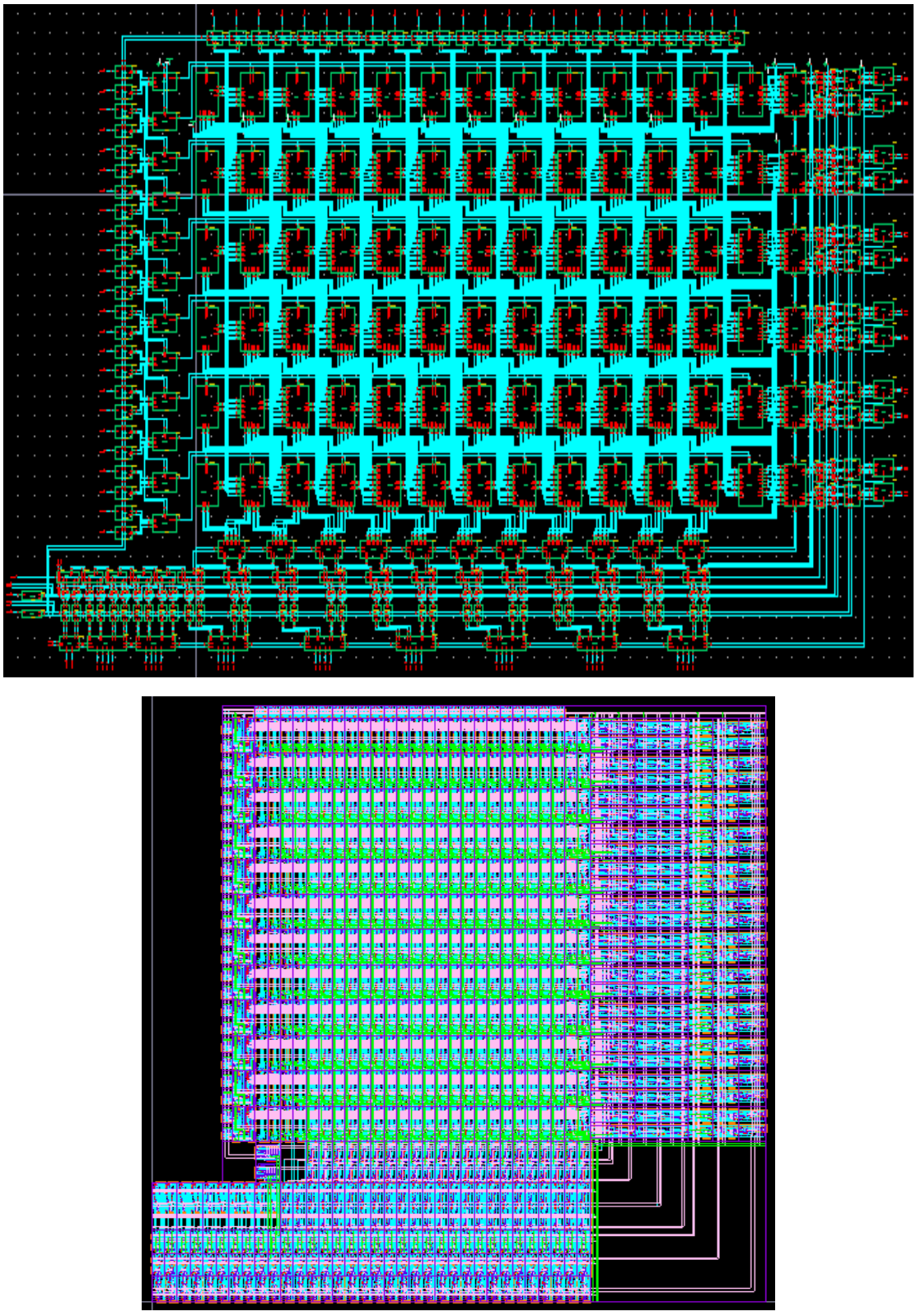


Figure 2-20: Schematic and Layout of 12x12b complex MAC

2.3.2.1. Cell Description

The following are the contents of the cells that are used in the complex MAC. Same as the multiplier design. The routing is included in each cell so that by tiling these cells together, one gets the entire complex multiplier (see Figure 2-20)

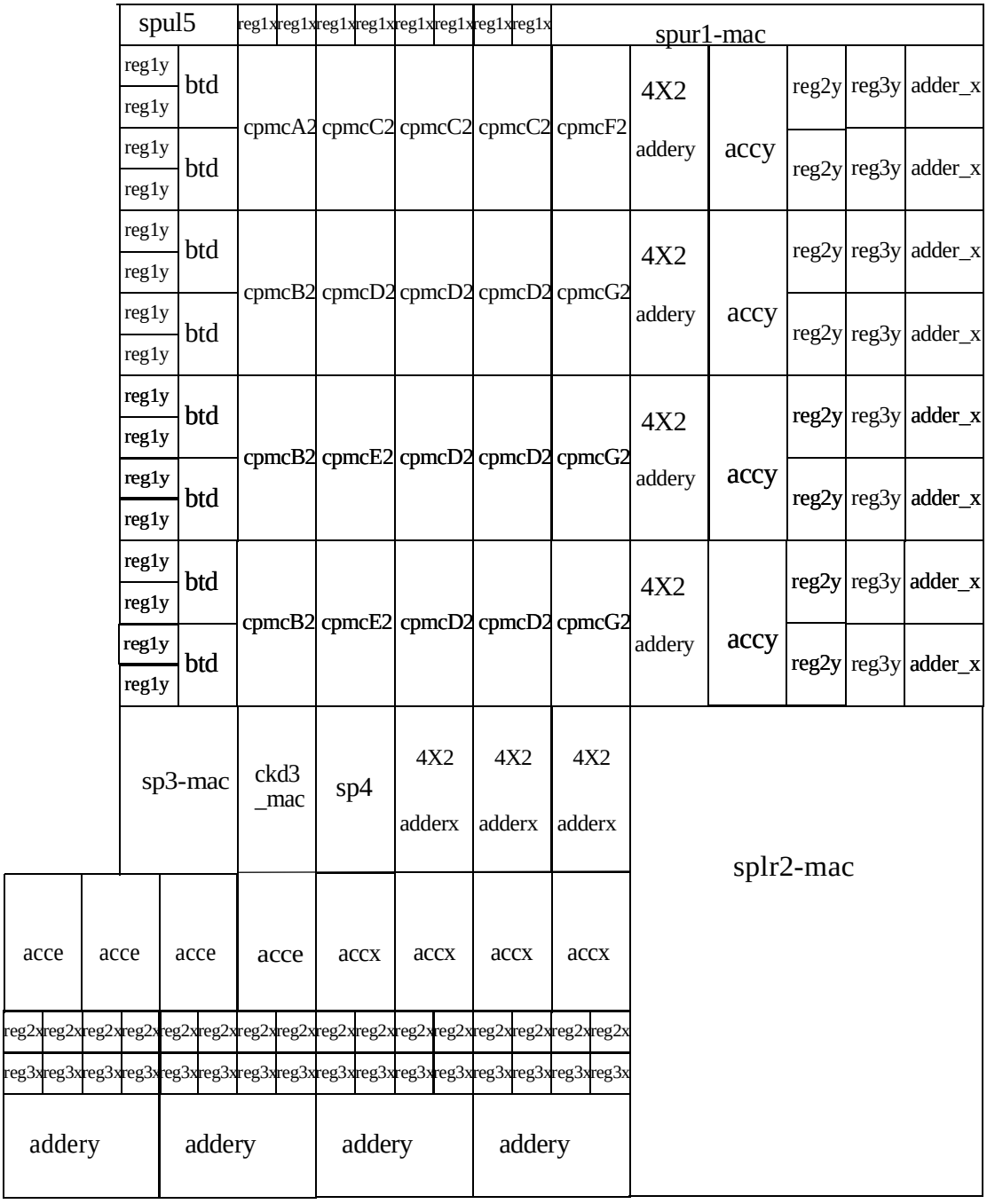


Figure 2-21: Cells of 4x8 Complex Multiplier Accumulator

reg1y, reg1x, reg3x -> one register.

reg 2x, reg3y -> two registers.

reg2y -> four registers.

adder_x -> four adders.

addery, adderx-mac -> two adders.

4x2adderx -> two 4:2 compressors.

4x2addery -> four 4:2 compressors.

accx, accy -> two 4:2 compressors with reset.

acce -> one accumulator adder for sign extension.

btd -> one booth encoder.

ckd3_mac -> two clock line buffer.

cpmc[A2-G2] -> four ary[a-g]s.

spur1-mac, splr2-mac, sp3-mac, sp4, spul5 --> wiring cells.

2.3.2.2 Circuit Design

Two new circuits are design for the MAC. One is the 4:2 compressor with reset, another is the accumulator adder for the sign extension.

4:2 Compressor with Reset

We need to reset the value store in the register B when MAC accumulates 15 times. The 4:2 compressor with reset is shown in Figure 2-18. Both In1 and In2 signals are from the register B, and In3 and In4 are from the multiplier array. When the reset signal goes low the value stored in register B will be dumped

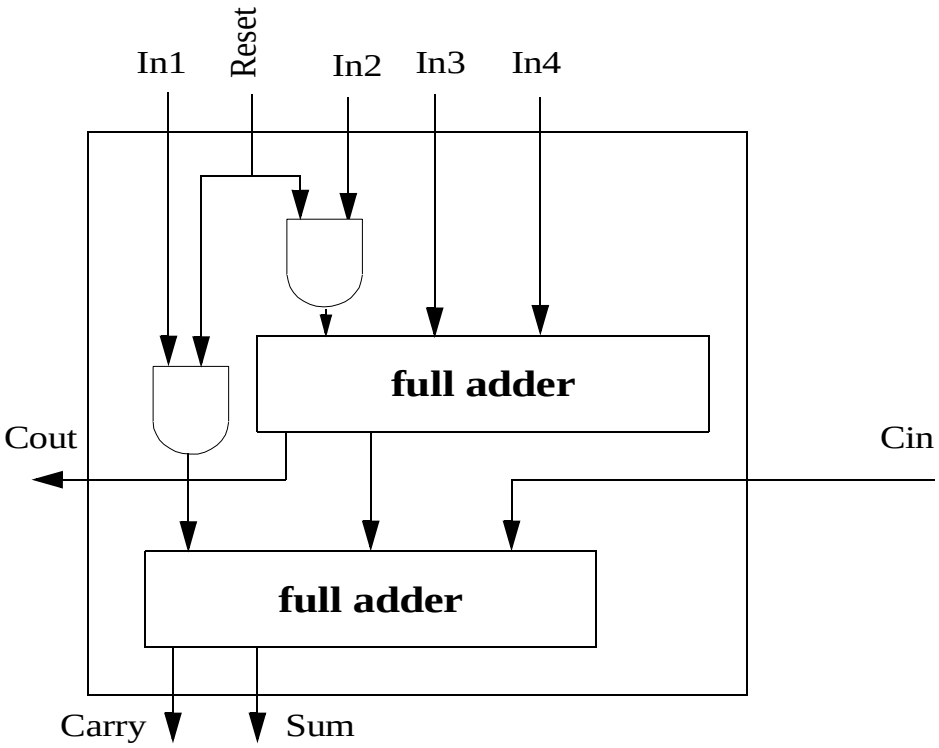


Figure 2-22: A 4:2 Compressor with Reset

Accumulator Adder for the Sign Extension

For a 12 bits by 12 bits complex MAC, after it accumulates 15 times the output has 29 bits. So we need extend the sign bit when we do the accumulation. The accumulator adder for the sign extension is shown in figure 2-19. It has 4 inputs and three outputs. Cin is the carry out from the adjacent 4:2 compressor and In1 and In2 are from the register B.

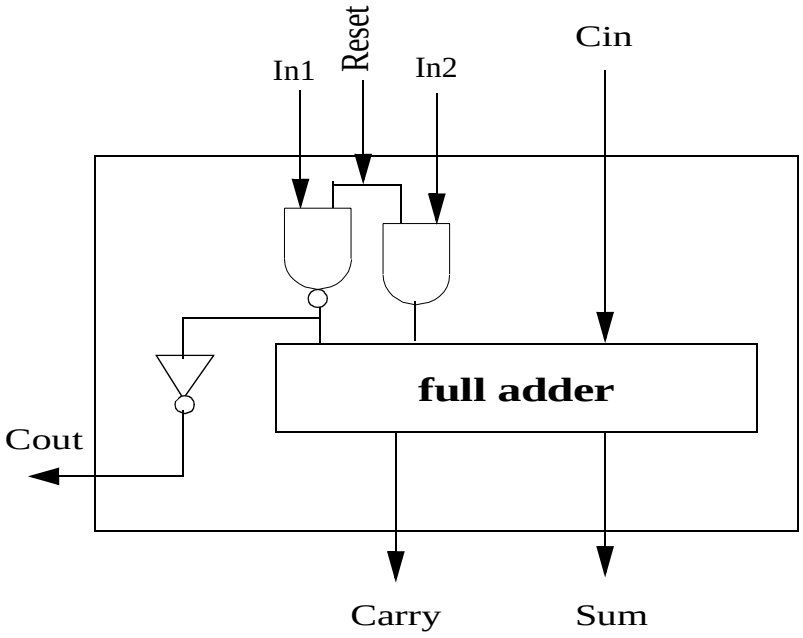


Figure 2-23 Accumulator adder for sign extension

2.4 Performance

The simulation of the multiplier propagation delay was done with HSpice and PathMill. The power dissipation was done with PowerMill. (All netlists were extracted from the lay-out).

Table 2-3. Area

	X (um)	Y (um)	area (mm^2)
16x16b Integer Multiplier	250.6	309.65	0.078
12x12b Complex Multiplier	383.35	474.6	0.182
12x12b Complex MAC	505	538.4	0.272

Table 2-4. Power Consumption (1V, 25MHz)

16x16b Integer Multiplier	0.26 mW

Table 2-4. Power Consumption (1V, 25MHz)

12x12b Complex Multiplier	0.623mW
12x12b Complex MAC	0.931mW

Table 2-5. Propagation Delay

	First Stage	Second Stage
16x16b Integer Multiplier	16.5ns	23.5ns
12x12b Complex Multiplier	9.5ns	20.7ns
12x12b Complex MAC	13.3ns	27.0ns

2.5 Power Reduction via technology Improvements

Initially, the 16x16b integer multiplier was designed with 0.6 um CMOS technology and had a power consumption of 1.86 mW at 1.5 V. In shifting from 0.6 to 0.25 um technology, the reduction of power consumption can be estimated [6]. The total capacitance reduced to 33% for the 0.25 um design. With this result the power can be estimated as 1.86mW x 0.33 = 0.6138 mW. Also the supply voltage drops from 1.5V to 1V. Therefore, the power consumption can be calculated as 0.6138 mW x $(1/1.5)^2$ = 0.2728 mW. This estimated value also conforms well with the actual measures power consumption (0.26 mW).

3 Parameterized Cells

3.1. Pcell Fundamentals

Pcell (Parameterized Cell) is a CAD tool for parameterized design. The pcell we create is called a master which is a combination of the leaf cell layouts and the parameters we assign to those cells. After we assign the parameters to the leaf cells, we need to compile the master. The compiler will translate the master to the form of a SKILL procedure and stored in the database (SKILL is the command language of the cadence environment).The design flow of pcell is shown in Figure 3-1

.

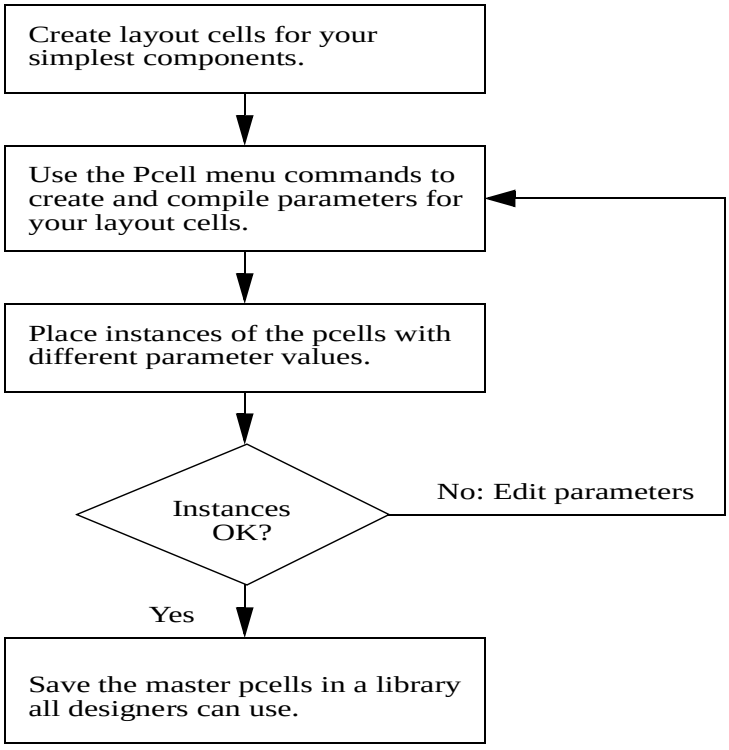


Figure 3-1: Pcell Design Flow

3.2 Design by abutment

In these designs, all the connections between the leaf cells are achieved by abutment. The big advantage of abutment is that abutment can reduce parasitic effects and improves lay-out density. However it has big advantage too. Figure 3-2 shown a hierarchical layout containing two instances of two leaf cells A and B. The pins are shown as squares. Due to the abutting requirements between the pins on common boundary, the relative position of these pins in the two cells should be the same. Determining these pin positions is impor-tant because it constrains the cell design. Once the hierarchy has been built, it is very tedious to make any changes to one of the leaf cell which would require its size or the position of one of the pins or ports to change. The change in the pin position in one of the leaf cells would cause a rippling effect in other cells. Also, each cell must save space between internal layouts and the bounding box to prevent design rule violations during abutment.

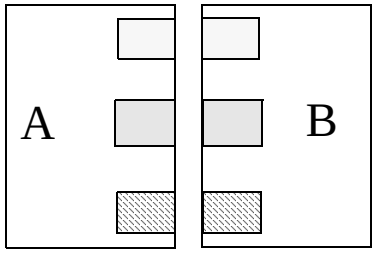


Figure 3-2: Abutment Requirements

3.3 Tutorial : Creating a pcell for the complex multiplier

3.3.1 Starting Pcell

1. Open the layout editor and create a new file.

2. To add the Pcell menu to the banner menu.

⇨ Select Tools → Pcells

*To make it easy to understand, the floor plan of a 12x12b complex multiplier is shown in Figure 3-3.

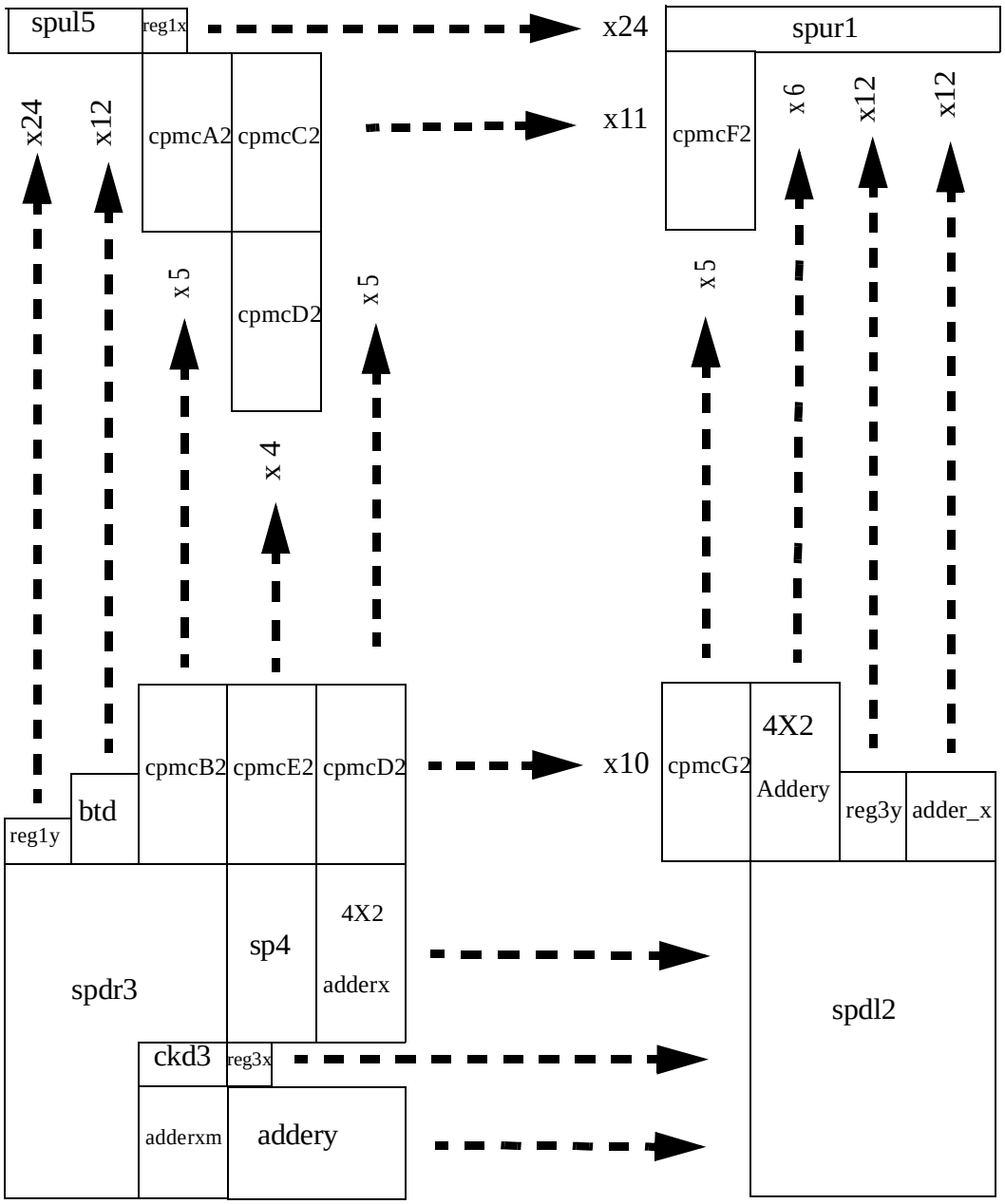
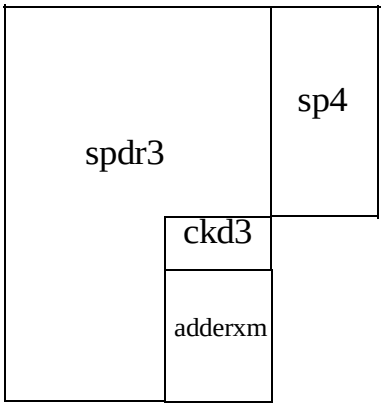


Figure 3-3: 12 bit x 12 bit Complex Multiplier Floor Plan

3.3.2 Place leaf cells

1. Start from the lower left corner, we place the leaf cells which are always necessary. In the Library Browser, click on hgxmpy -> hgspdr3, hgsp4, hgckd3, adderxm. Notice that pins are abutted properly.



3.3.3 Repetition Commands

Repetition commands repeats cells in the x direction, y direction, or both directions.

There are three useful repetition commands:

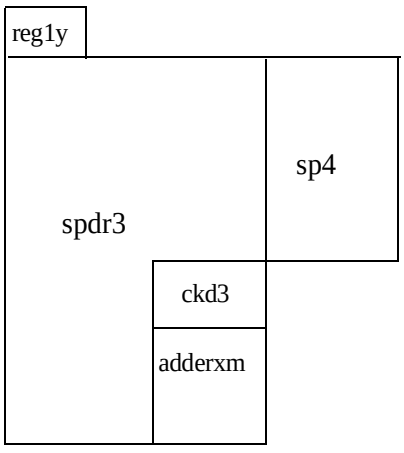
*Repeat in X, which defines cells to be repeated horizontally.

*Repeat in Y, which defines cells to be repeated vertically.

*Repeat in X and Y, which defines cells to be repeated both horizontally and vertically.

Now, we want the reg1y to be extendible depends on the parameter (multiplier operand).

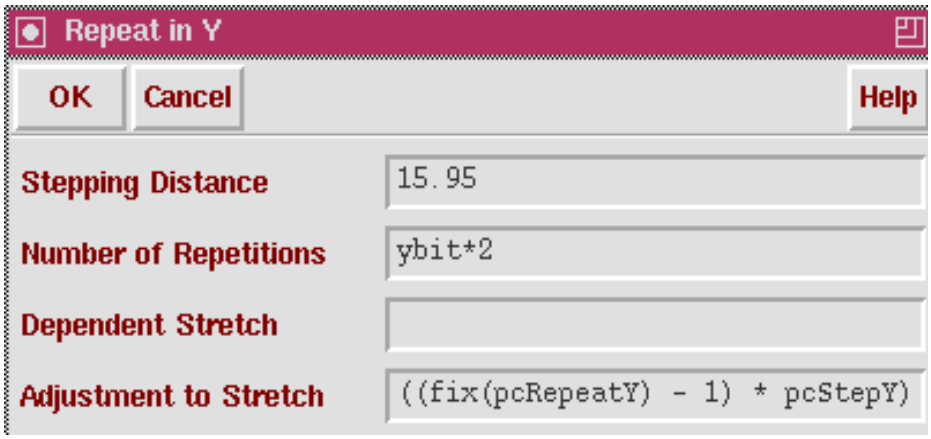
1. Place the reg1y cell. The pcell window should look like following:



2. Preselect the reg1y cell then start the command.

select Pcell → Repetition → Repeat in Y

The Repeat in Y dialog box should look similar to the following:



*if you did not preselect cells, the program prompts you to point to the cell you want to repeat.

Stepping Distance is the centerline-to-centerline distance you want between repeated cells.Type “15.95” in the Stepping Distance box. Because the height of regly cell is 15.95um. By default, repetition takes place in a positive direction (upward or to the right). To repeat in a negative direction (downward or to the left), you must use a negative number for the stepping distance. If you didn’t specify any value, the default value is zero.

Number of Repetitions is the number of times you want the specified objects repeated. You can use a number or an expression that is dependent on other parameters of the pcell.
The name must begin with an alphabetical character and cannot exceed 32 characters. Type (ybit*2) in the Number of Repetitions box. The input is a complex number so we need multiply by two.

Dependent Stretch is the name of the previously defined stretch control line. By default, stretching takes place before repetition. If you specify a dependent stretch control line, stretch takes place after repetition. About stretch control line, we will discuss it later.

Adjustment to stretch is the amount the program adjusts the reference dimension of the stretch parameter. We usually do not have to change it. The default value is ((fix (pcRepeatY-1)*pcStepY.

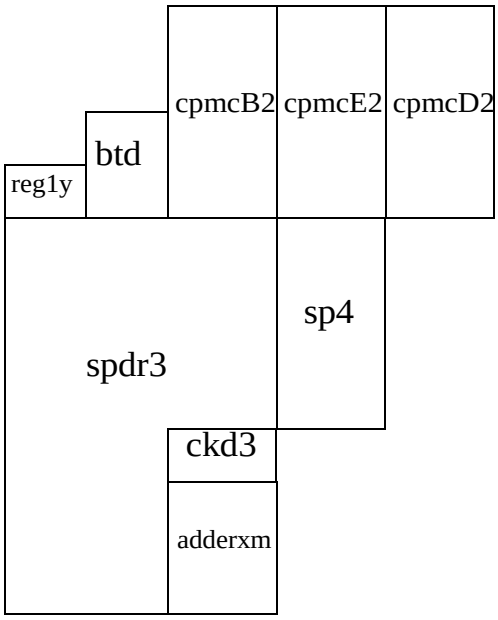
*fix() is a SKILL function, it runs the number down to the nearest integer.
*pcRepeatY is the number of vertical repetitions.
*pcStepY is the vertical stepping distance.

3. Place the btd cell. Repeat step 2.
   Type 31.9 into the Stepping Distance box. The height of btd cell is 31.9 um.
   Type ybit into the Number of Repetition box.

4. Place cpmcB2 cell. Repeat step2.
   Type 63.8 into the Stepping Distance box. The height of cpmcB2 cell is 63.8 um.
   Type ((ybit/2)-1) into the Number of Repetition box.

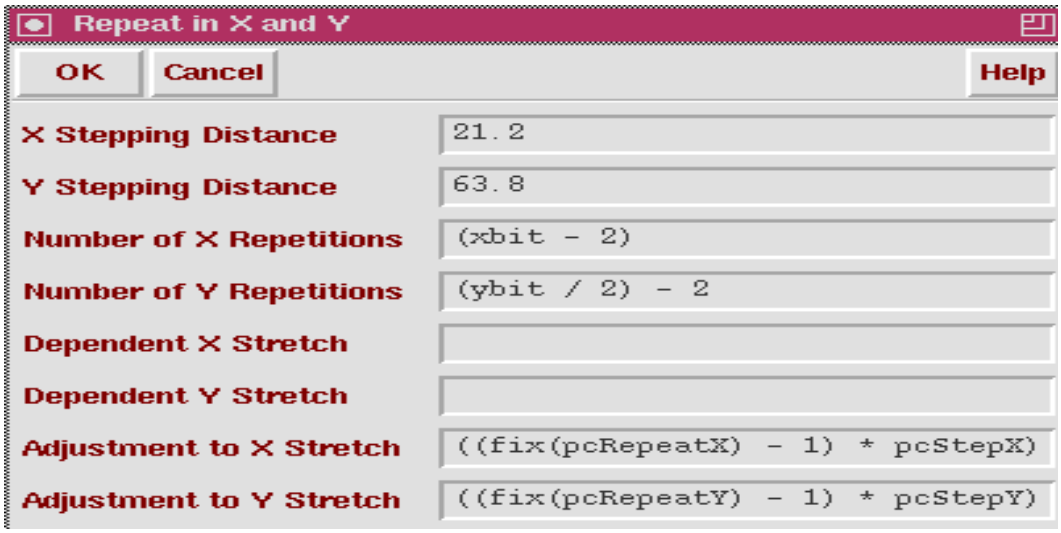
5. Place cpmcE2 cell. Repeat step 2.
   Type 63.8 into the Stepping Distance box. The height of cpmcE2 cell is 63.8 um.
   Type ((ybit/2)-2) into the Number of Repetition box.

6. Place cpmcD2 cell then start the command



Select Pcell -> Repetition -> Repeat in X and Y

The Repeat in X and Y dialog box should look similar to the following:



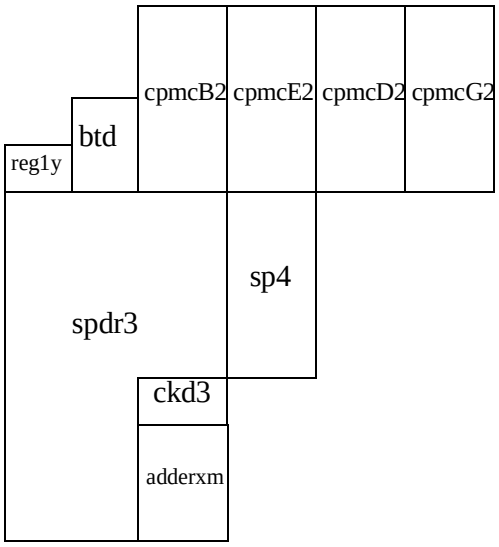
Type 21.2 into the X Stepping Distance box. The width of cpmcD2 cell is 21.2 um.

Type 63.8 into the Y Stepping Distance box. The height of cpmcD2 cell is 63.8 um.

Type (xbit-2) into the Number of X Repetitions box.

Type ((ybit/2)-1) into the Number of Y Repetitions box.

7. Place cpmcG2 cell.



Repeat step2.

Type 63.8 into the Stepping Distance box. The height of cpmcG2 cell is 63.8 um.

Type ((ybit/2)-1) into the Number of Repetition box.

8. Place 4X2addery cell. Repeat step 2.

Type 63.8 into the Stepping Distance box. The height of 4X2addery cell is 63.8 um.

Type (ybit/4) into the Number of Repetition box.

9. Place reg3y cell. Repeat step 2.

Type 31.9 into the Stepping Distance box. The height of reg3y cell is 31.9 um.

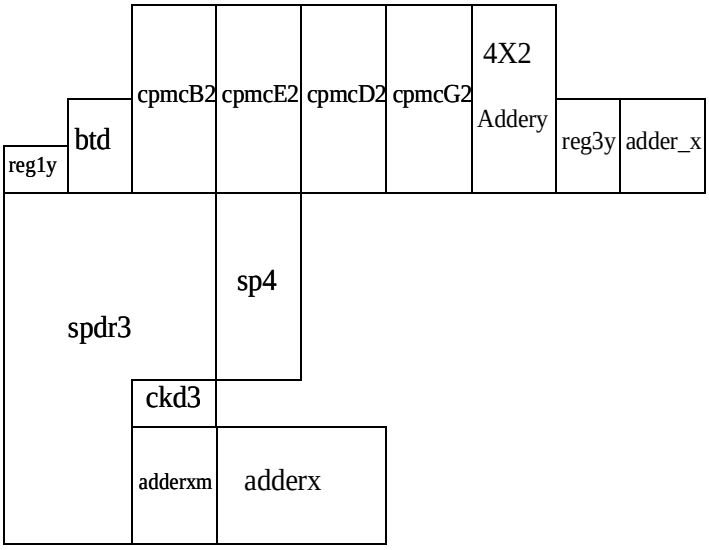
Type ybit into the Number of Repetition box.

10. Place adder_x cell. Repeat step 2.

Type 31.9 into the Stepping Distance box. The height of adder_x cell is 31.9 um.

Type ybit into the Number of Repetition box.

12. Place adderx cell, The pcell window should look like following now.



13. Preselect the adderx cell then start the command.

    Select Pcell -> Repetition -> Repeat in X...

    Type 42.4 into the Stepping Distance box. The width of the adderx cell is 42.4um.

    Type (xbit/2) into the Number of Repetition box.

14. Place reg3x cell. Repeat step 13.

    Type 10.6 into the Stepping Distance box. The width of the reg3x cell is 10.6um.

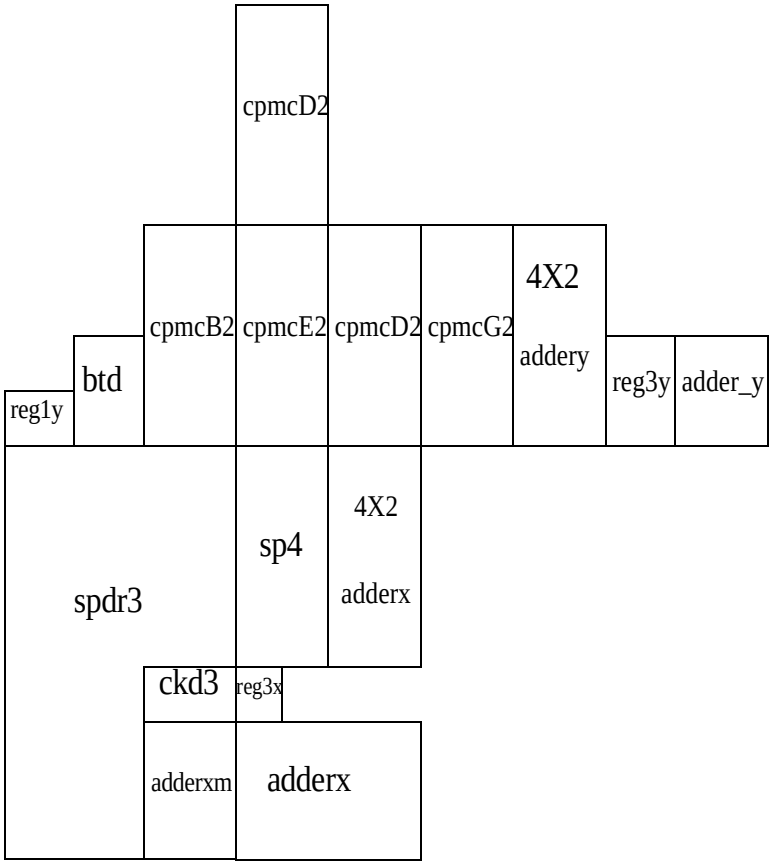
    Type (xbit*2) into the Number of Repetition box.

15. Place 4X2adderx cell. Repeat step 13.

    Type 21.2 into the Stepping Distance box. The width of the 4X2adderx cell is 21.2um.

    Type (xbit-1) into the Number of Repetition box.

16. Place the second cpmcD2 cell. In order to distinguish this cpmcD2 from the previous one. I call this cpmcD2 cell to second-cpmcD2, the previous one is first-cpmcD2.



The second-cpmcD2 is always on the top of cpmcE2 and it doesn’t need the repetition. We don’t need assign parameter to it now.

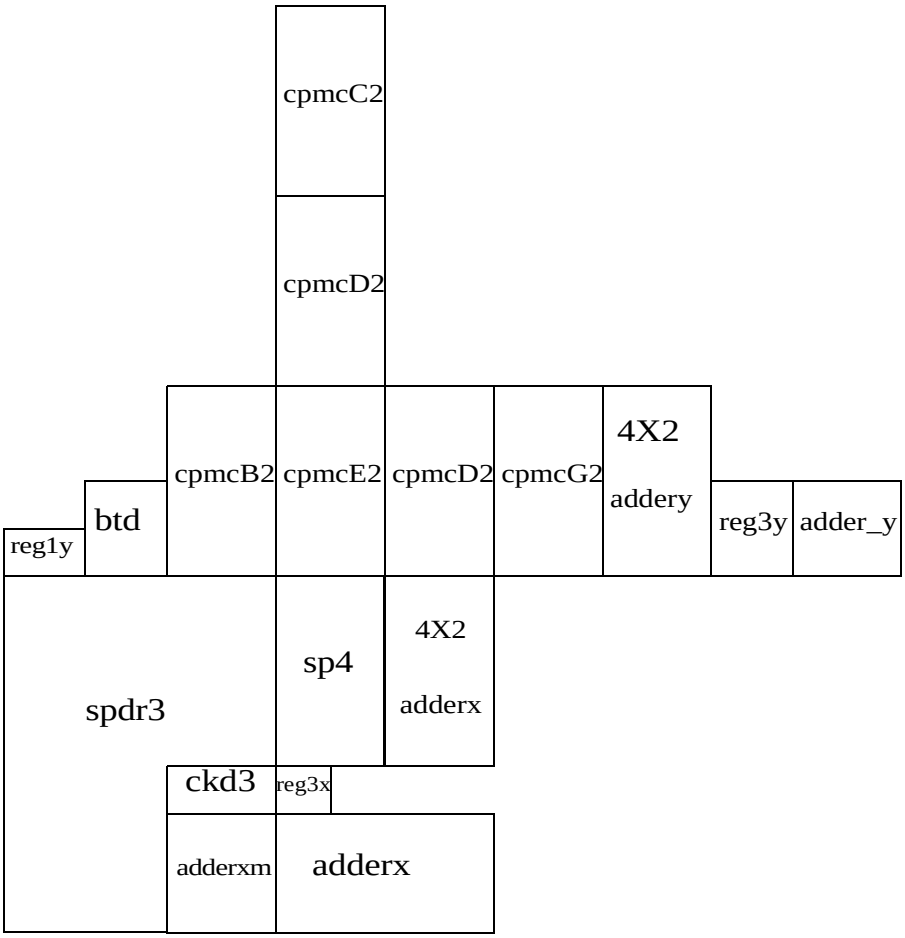
17. Place cpmcC2 cell. It always on the top the second cpmcD2 cell.

    Repeat step 13.

    Type 21.2 into the Stepping Distance box. The width of the cpmcC2 cell is 21.2 um.

    Type xbit-1 into the Number of Repetition box.

.



18. Place spdl2 cell. This cell is necessary and it always at the lower right corner.

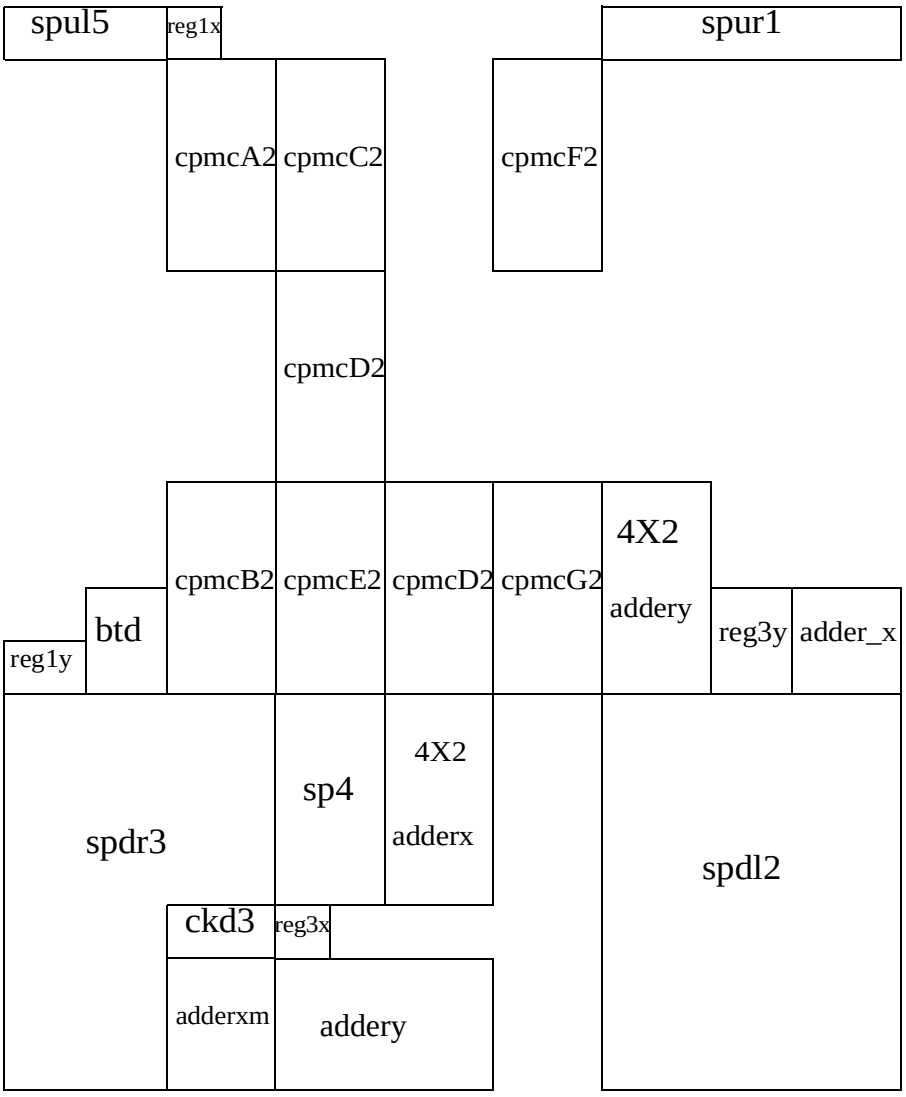
19. Place cpmcA2 cell. This cell is always on the left side of cpmcC2 cell.

20. Place reg1x cell. Repeat step 13.
    Type 10.6 into the Stepping Distance box. The width of the reg1x cell is 10.6 um.
    Type (xbit*2) into the Number of Repetition box.

21. Place spul5 cell. This cell is always at the upper left corner.

22. Place spurl cell. This cell is always at the upper right corner.

23. Place cpmcF2 cell.



Now, we place all the cells we need for a complex multiplier**.**

3.3.4 Dependent Stretch Control Line

Stretch control lines determine where to begin the stretch and which direction to stretch. Because the size of our leaf cells is already fixed, we don't need to use stretch commands to stretch our leaf cells. However, we need the stretch control line to stretch the space for the extended leaf cells. Otherwise some leaf cells will overlap. For example, when the first-cpmcD2 cell repeats in the X direction, then it will overlap those cells (cpmcG2,

reg3y...) on its right hand side. And the size of space is different for different repetition parameters. Thus, we need set the stretch control line dependent on the repetition parameters. When the repetition parameters are given, the numbers of repetition will control the amount of stretch.

3.3.4.1 Making the Stretch Layer Valid

Before using a stretch control line, we must make the stretch layer valid.

To make the stretch layer valid

1. In the LSW (Layer and Selection Window), select Edit -> Set Valid Layers.

The Set Valid Layers form appears.

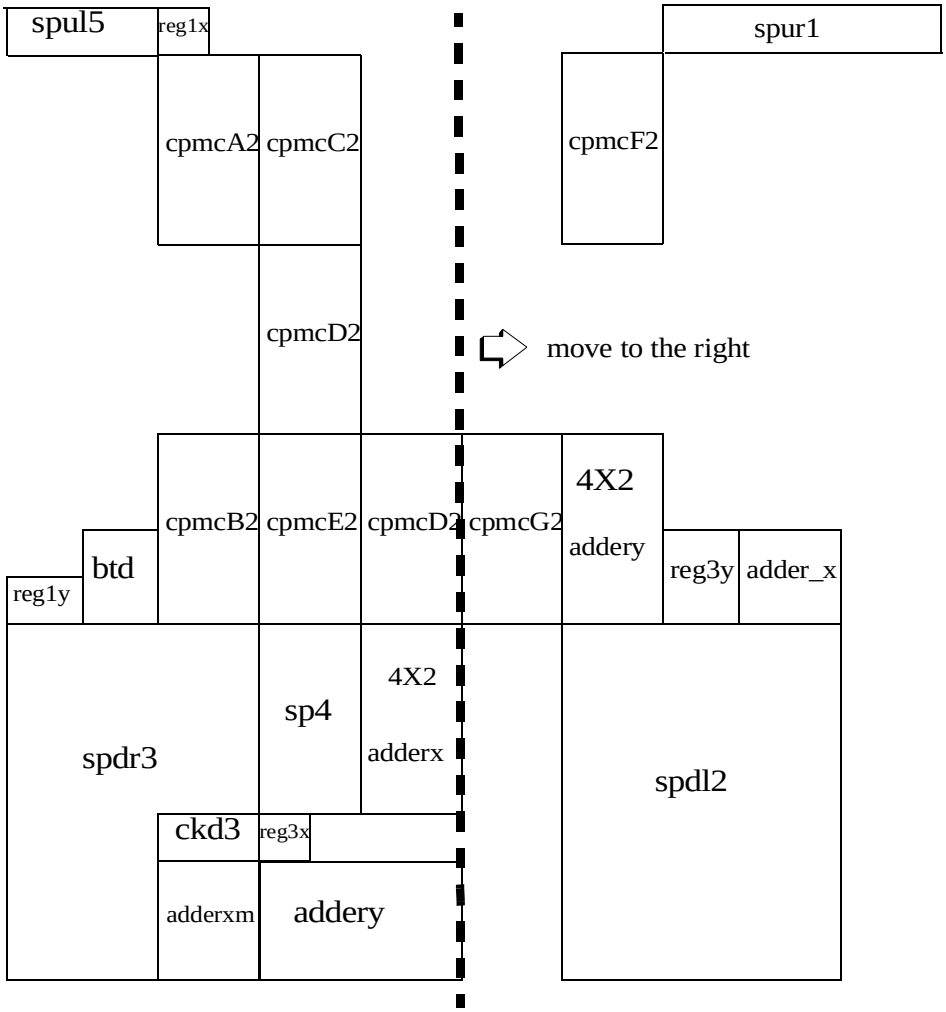
2. In the Set Valid Layers form, click the box to the right of stretch dg.

3. Click OK to close the form.

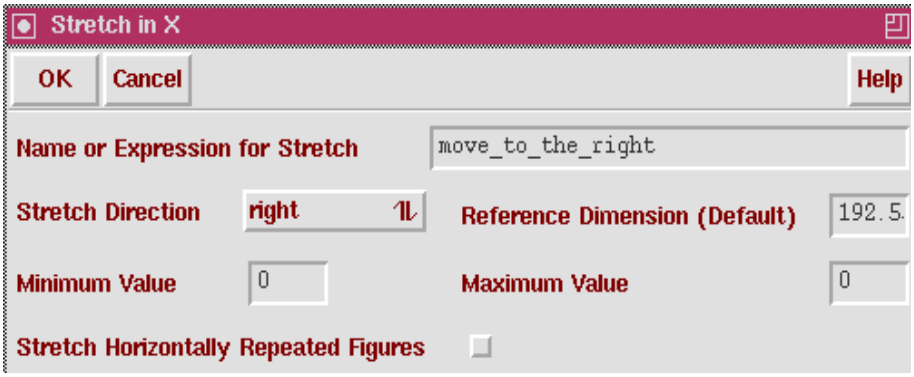
3.3.4.2 Stretch Commands

4. Select Pcell -> Stretch -> Stretch in X.

5. Draw a vertical stretch line through the first-cpmcD2 cell.



The Stretch in X dialog box appears.



Name or Expression for Stretch is a parameter name or SKILL expression.

Reference Dimension(Default) is the default value for the stretch control line. When you place a pcell, the system subtracts the reference dimension from the stretch value you specify to determine the distance to stretch.

Stretch Direction is the direction you want objects to stretch.

Stretch Horizontally Repeated Figures stretch cells marked for repetition in the X direction. By default, cells in repetition groups are not stretch (the button is not filled). That is why the stretch control line can through the cpmcD2 cell. If the button is on. horizontally repeated cells will stretch.

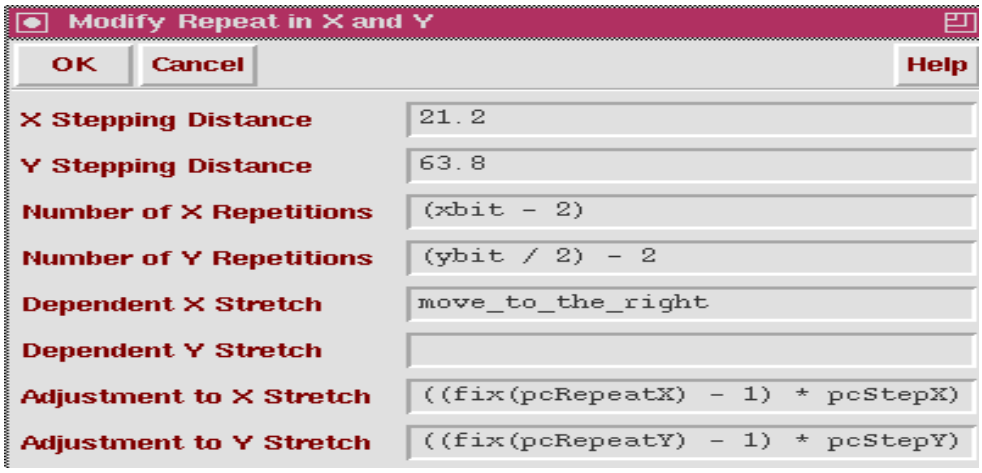
Minimum Value specifies the smallest value you can use to stretch this pcell. If you enter a value smaller than the minimum, the Pcell program uses the minimum value.

Maximum Value specifies the largest value you can enter when stretching this pcell.

6. Now, we need to modify the parameters of the first-cpmcD2 cell. To assign the first-cpmcD2 cell's repetition parameters to control the stretch control line.

   ⇨ select Pcell -> Repetition -> Modify

   Double click the first pcmcD2 cell. The Modify Repeat in X and Y dialog box appear.

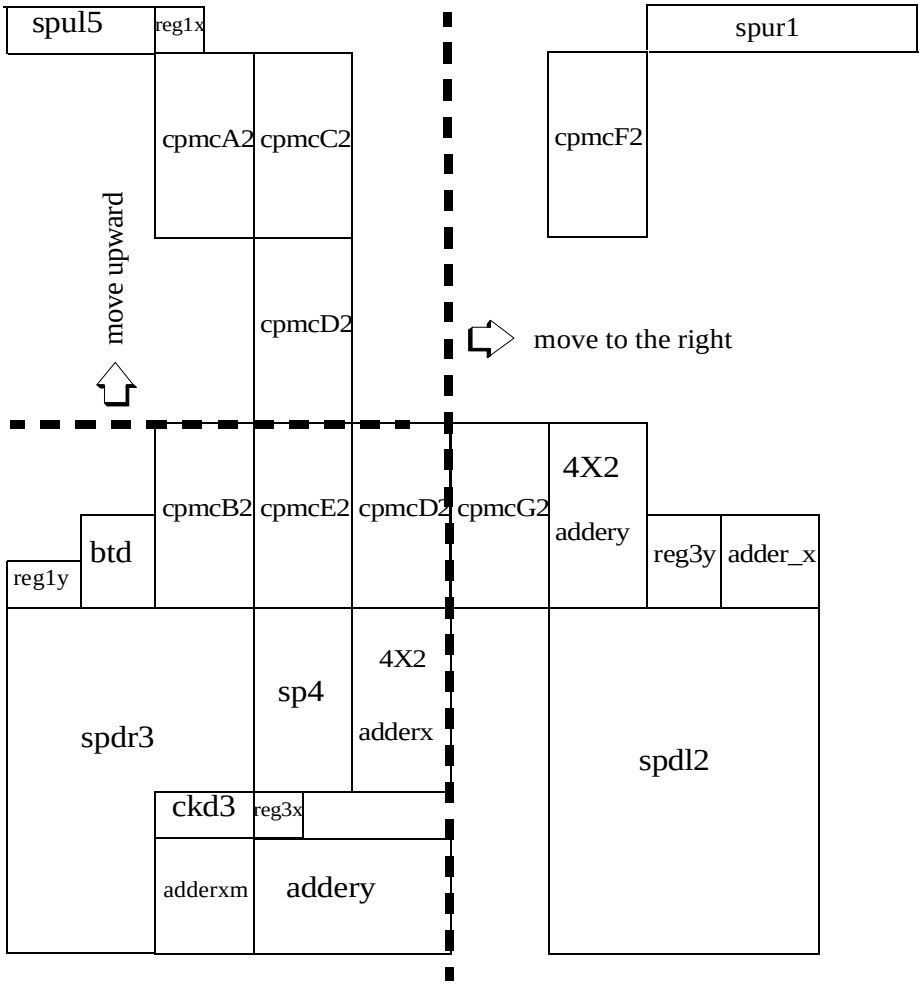


   Type move_to_the _right in the Dependent X stretch box.

7. Select Pcell -> Stretch -> Stretch in Y.

8. Draw a horizontal stretch line through the cpmcE2 cells.

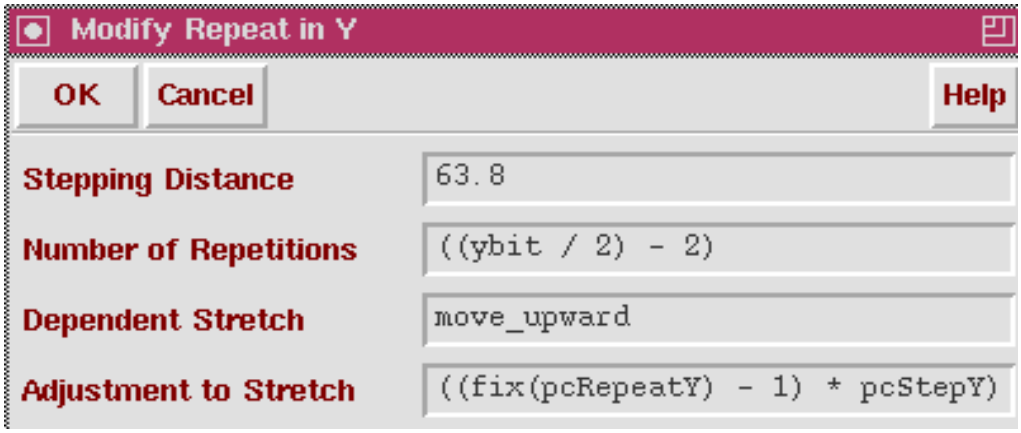
   Type move_upward in the Dependent Y Stretch box.



9. Now, we want the repetition parameter of cpmcE2 cell to control the second stretch control line.

   ⇨ select Pcell -> Repetition -> Modify

   Double click the pcmcE2 cell. The Modify Repeat in Y dialog box appear.



Type move_upward in the Dependent Y Stretch box.

3.3.5 Conditional Inclusion Commands

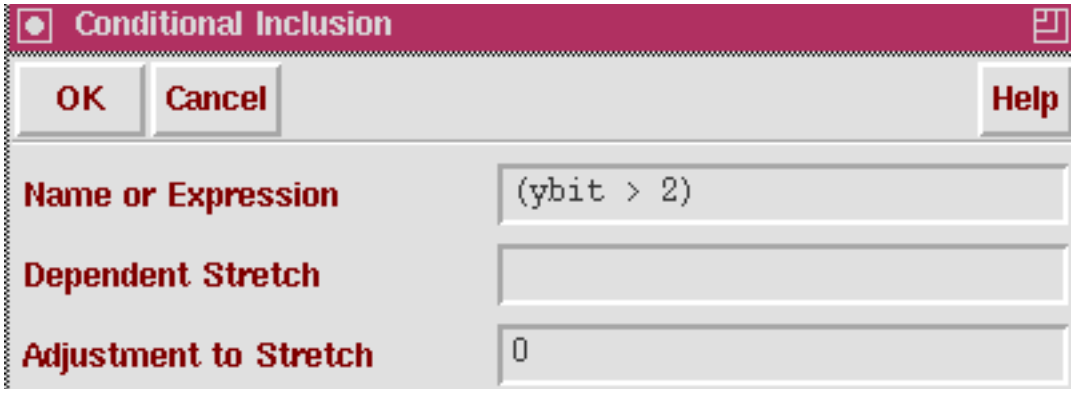
Conditional Inclusion commands set a parameter that includes or excludes objects depending on the conditions you set. These commands can be used in conjunction with stretch or repetition commands.

Now, the smallest complex multiplier that can be generated is a 4x2. Because no matter what size it is, at least we have cpmcC2, cpmcD2 in the same column. We can use conditional inclusion commands to exclude cpmcD2 when ybit is less than four. We don't have to worry about cpmcE2. Because the Repeat in X parameter of cpmcE2 is zero when ybit. is four. When the parameter is zero, it means the cell doesn't exist.

1. Preselect the second cpmcD2 cells.(the one between cpmcC2 and cpmcE2). Start the command.

   ⇨ Select Pcell -> Conditional Inclusion -> Define.

   The Conditional Inclusion window appears.



Name or Expression is the definition of the parameter or SKILL expression. When you place an instance of the pcell, the Pcell program evaluates this expression to determine whether to include the specified objects.

Dependent Stretch is the name of a previously defined stretch control line. We can set this stretch control line to be dependent on conditionally included cells.

Adjustment to Stretch is the amount the stretch control line stretches or compresses when the conditional cells are not included. It can be a positive or negative value.

2. Type in (ybit > 2) in the Name or Expression box.

3. click OK.

Now, we have completed all the parameters.

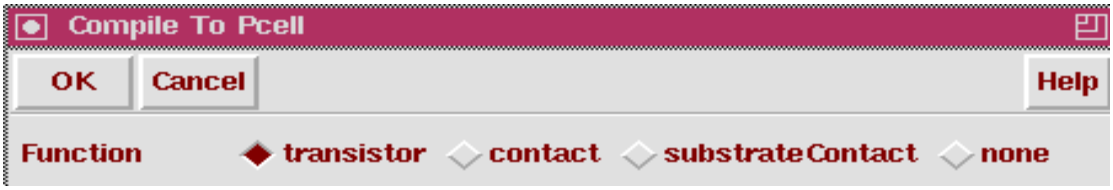
3.3.6 Compile and Testing the Pcell

Create pcells is often an iterative process. It is not easy to set all the parameters correctly at the first time. However, we can easily go back and add parameters or make corrections.

3.3.6.1 Compiling the Pcell

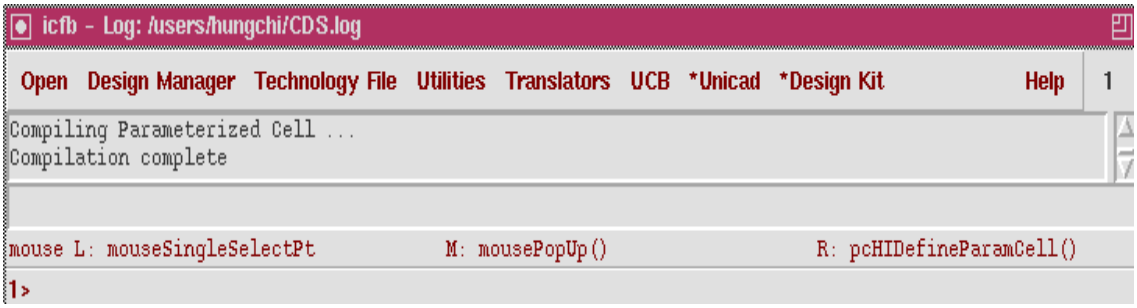
1. Pcell -> Compile -> To Pcell.

The Compile To Pcell form appears. The first time we compile a parameterized cell, we need to classify it as transistor, contact, substrate contact, or none. These are the classifications that Cadence virtuoso Compactor uses.



2. Select transistor, click ok.

We can see the following message in the CIW (Command Interpreter Window).



If the output in the CIW shows any errors, we go back to fix it then recompile the pcell again.

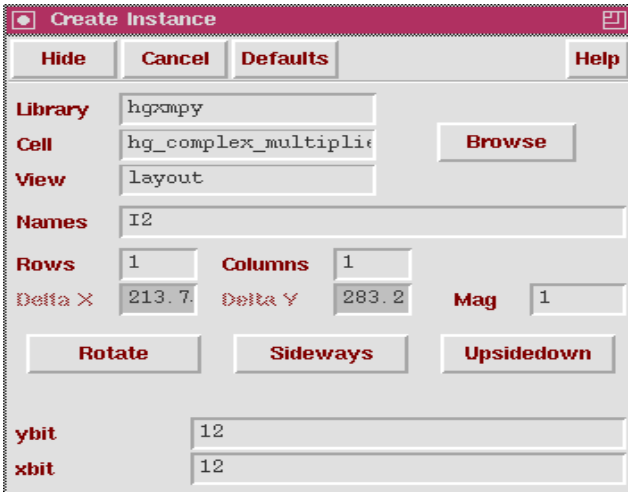
3. Save the pcell. Select Design -> Save.

3.3.6.2 Testing the Pcell

1. Open a new file.

2. Open the Create Instance form, by pressing i in the new window.

The Create Instance form appears.



3. In the Create Instance form, enter hgxmpy in the Library box and hg_complex_multiplier-master in the Cell box.

4. Press Tab to display the pcell parameters.
   The parameters of complex multiplier are displayed at the bottom of the Create Instance form. The values shown are the defaults.

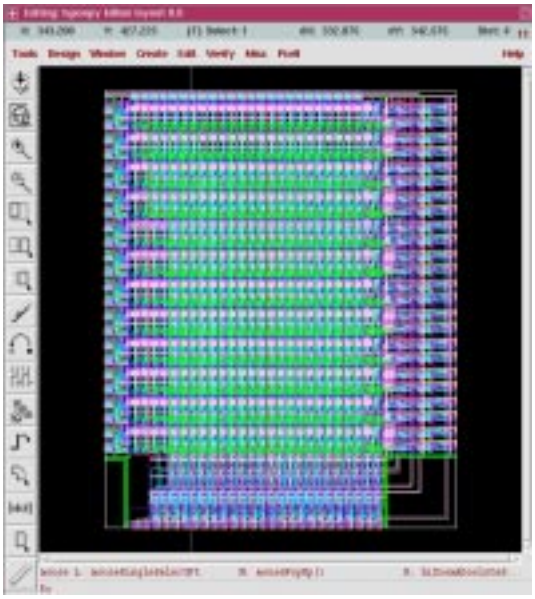
5. Change ybit to 12.

6. Change xbit to 12.

7. Place the complex multiplier, click left mouse bottom in the new window.

8. Then click cancel in the Create Instance Form.

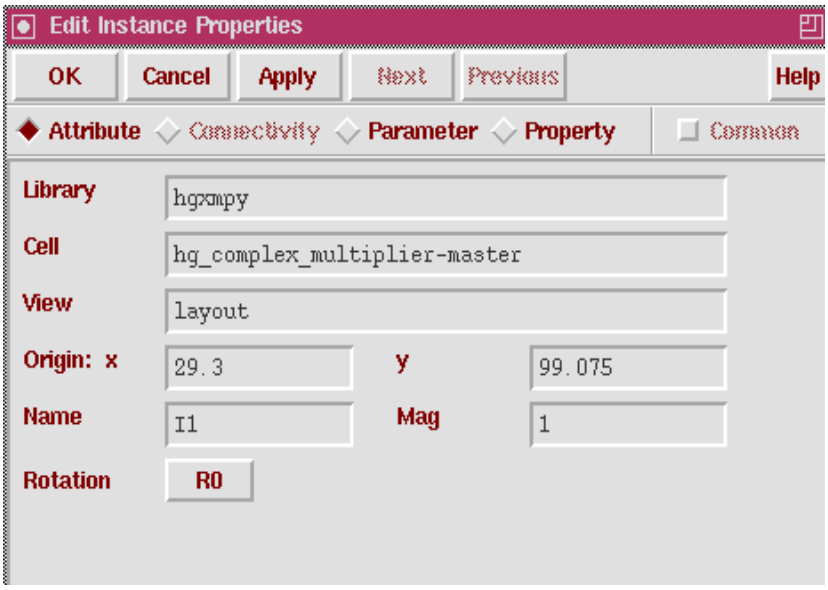
9. Press f in the new window to fit the 12x12 complex multiplier in the window.



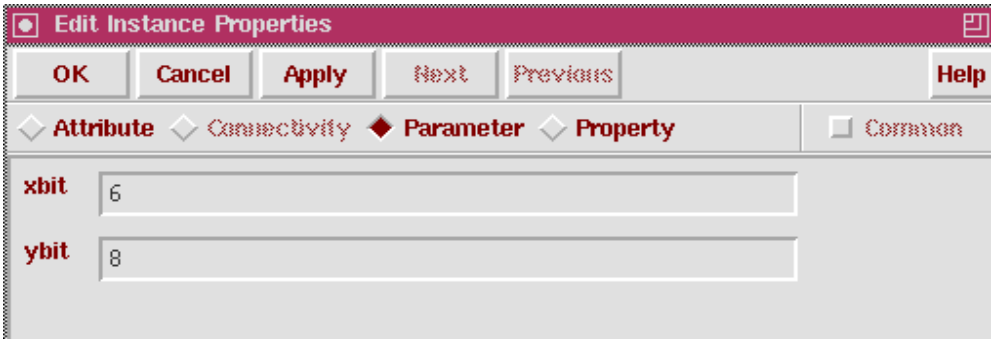
Now we change the parameter values after we have placed the complex multiplier

10. Select the complex multiplier, then press q.

The Edit Instance Properties windows appears.



12. In the Edit Properties Form, click on the Parameter button.

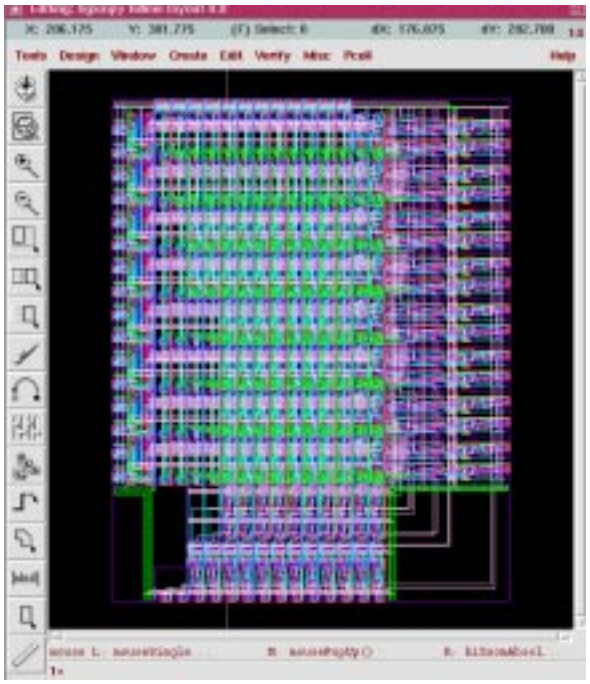


13. Change xbit to 6.

14. Change ybit to 8.

15. Click OK.

    The complex multiplier will change to 6X8 complex multiplier.



16. If there is no error, we have completed the pcell for the complex multiplier.

3.4 Summary

In this chapter I used three most useful commands (Repetition Commands, Stretch Command, and Conditional Inclusion Commands). There are still more commands you can learn from the Cadence menu.

4 Conclusions

Three pcells have been created for the pcell library. The first and the second are integer number and complex number multiplier. They are capable of generating integer and complex number multipliers of size 2x2 or larger. Both the multiplier and multiplicand vectors can be increased from 2 to any larger number in increment of 2. The third pcell is a complex number multiplier accumulator. It is capable of generating integer and complex number multipliers of size 4x2 or larger. Both the multiplier and multiplicand vectors can be increased to any larger number in increment of 2. In addition to these features, this complex MAC has parameterizable accumulation times. This feature make this complex MAC can be apply to more different projects.

There is still one thing left to be done. From the performance (section 2.4). we can see the critical path has changed from the first stage to the second stage in 0.25 um technology. Although, it meets our clock speed (25MHz or 40 ns clock period). We should still add another parameter to include or exclude the carry look ahead cell in the second stage, to reduce the delay of the ripple carry adder. This feature will make these pcells be applied to more different projects.

Bibliography

[1] J. M. Rabaey, Digital Integrated Circuits. Prentice Hall, 1996.

[2] J. M. Rabaey, M. Pedram. Low Power Design Methodologies. Kluwer Academic Publishers, 1996.

[2] M. Izumikawa, et al., “A 0.25-um CMOS 0.9-V 100-MHz DSP Core”, IEEE Journal of Solid-State Circuits vol.32, no. 1, January 1997.

[3] G. Goto, et al ., “A 4.1-ns Compact 54x54-b Multiplier Utilizing Sign-Select Booth Encoders”, IEEE Journal of Solid-State Circuits vol.32, no. 11, November 1997.

[4] G. Goto, et al ., “A 54x54-b Regularly Structured Tree Multiplier”, IEEE Journal of Solid-State Circuits vol.27, no. 9, September 1992.

[5] V. G. Moshnyaga, K. Tamaru. “A Comparative Study of Switching Activity Reduction Techniques for Design of Low-Power Multipliers. IEEE Journal of Solid-State Circuits vol.32, no. 11, November 1995.

[6] D. Carlson, et al., “A 667 MHz RISC Microprocessor Containing a 6.0ns 64b Integer Multiplier”, ISSCC, February 1998.

[7] Y. Hagihara, et al., “A 2.7ns 0.25um CMOS 54x54b Multiplier”, ISSCC, february 1998.

[8] F. F. Isluam, K. Tamaru. “High Speed Merged Array Multiplication”, Journal of VLSI Signal Processing, 1995.

[9] N. Zhang . “Implementation Issues in the Design of a CDMA Baseband Receiver” UCB MS Thesis, 1998.

[10] R. L. Maziasz, J. P. Hayes, Layout Minimization of CMOS Cells. Kluwer Academic Publishers, 1992.

[11] C. Bamji, R. Varadarajan, Leaf Cell And Hierarchical Compaction Techniques. Kluwer Academic Publishers, 1997.

[12] Skill Language User Guide. Cadence Design Systems, February 1997.

[13] Virtuoso Parameterized Cell Reference Manual, Cadence Design Systems, February 1997.

[14] Design Framework II SKILL Functions Reference, Cadence Design Systems, February 1997.