Arithmetic Devices

Part I: 16-bit Carry Select Adder

1. Theoretical overview

At the very least, most people expect computers to do some kind of arithmetic computation, and thus, most people expect computers to add. We're going to construct combinational logic circuits that perform binary addition. The adder is a logic element which computes the (n + 1)-bit sum of two n-bit numbers. We know that combinational logic circuits can't compute the outputs instantaneously. There is some delay between the time the inputs are sent to the circuit, and the time the output is computed. Let's say the delay is T units of time. Suppose you want to implement an n-bit ripple carry adder. How much total delay is there? Since an n-bit ripple carry adder consists of n adders, there will be a delay of **nT**. Why is there this much delay? After all, aren't the adders working in parallel? While the adders are working in parallel, carries must "ripple" their way from the least significant bit and work their way to the most significant bit. One of the possible approaches is Carry Select Adder (CSA). Carry select adders add much faster than ripple carry adders. The carry select adder generally consists of Ripple Carry Adders (RCA) and one multiplexer. Figure 1 shows a 16-bit carry select adder.

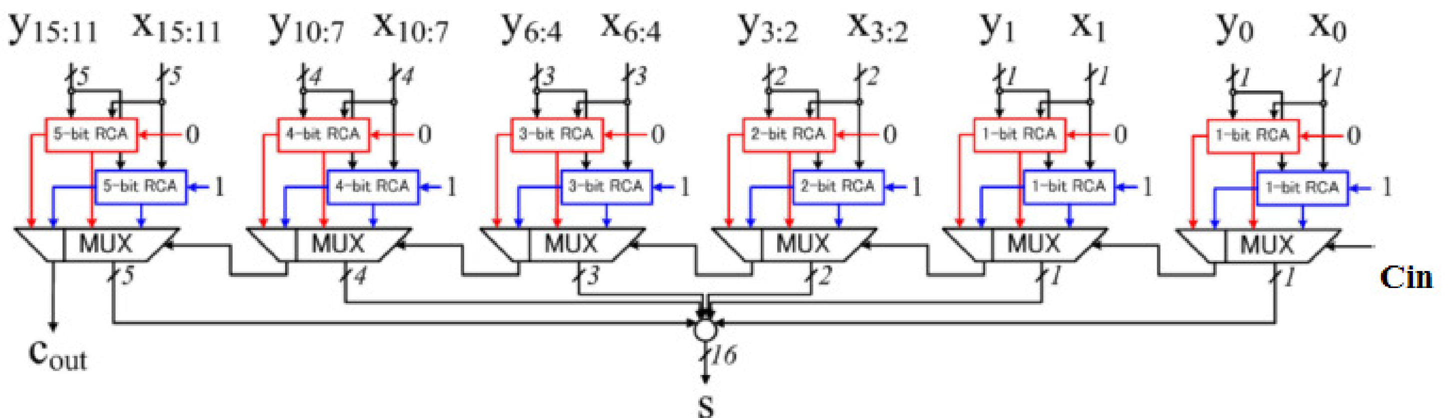

Figure 1: 16 bit Carry Select Adder

2. Implementation

2.1 Describe as independent **generic** component the RCA and MUX pair (RCAMP) by using **processes** (RTLl level).

2.2 Describe the full circuit by using RCAM component and FOR/IF **generate** statement (structural level).

2.3 Describe the RCA 16-bit and compare the hardware cost (number of logic elements) and performance (delay) of RCA with CSA. Use Altera Quartus II tool and Cyclone II FPGA for fulfillment this task.

Part II: 16 bit fast multiplier

פעולת כפל היא אחת מהפעולות הבסיסיות הנתמכות ברמת חומרה. חשיבות של מתן תמיכה מיטבית בפעולה זאת עולה עם מעבר מסיבי של אלגוריתמי DSP ליישומי חומרה, אשר בהם פעולת הכפל היא הפעולה המרכזית הקובעת ביצועי המערכת והצרכנית המרכזית של המשאבים.

להלן תיאור האלגוריתם היעיל המתבסס על זיכרון בו שמורות תוצאות הכפל. שליפת התוצאה מהזיכרון היא הפעולה הפשוטה והמתבקשת, אך המגבלה הגדולה של השימוש בגישה זו היא כמות הזיכרון הנדרשת. בעיה זו מודגמת באיור הבא:

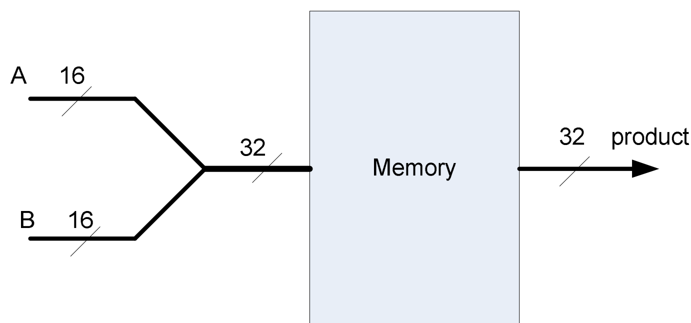

Figure 1

הנכפל A של 16 ביט והכופל B של 16 ביט מהווים כתובת לזיכרון המכיל תוצאות הכפל, גודל הזיכרון בדוגמה זו הוא: $total_memory_size=32 * 2^{(16+16)} = 2^{37}\ bit\ !!!$, מובן שדרך זו אינה מתאפשרת. הפתרון המתקבל מבחינת פשרה בין הגודל הזיכרון הנדרש לבין מספר פעולות החיבור של תוצאות הביניים. פתרון זה מציע חלוקה של הכופל והנכפל לתת-קבוצות. באיור הבא מוצג פתרון של חלוקת הארגומנטים של 16 ביט לתת-קבוצות של 8 ביט כל אחת.

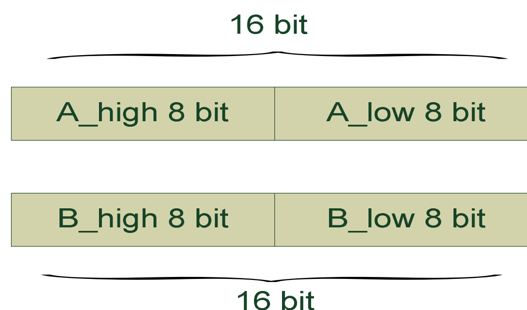

Figure 2

התוצאה תחושב באופן הבא:

$$product = a_low * b_low + ((a_high * b_low) << 8) + ((a_low * b_high) << 8) + ((a_high * b_high) << 16)$$

כל התוצאות הביניים נשלפות מהזיכרונות המחזיקים את תוצאות הכפל של תת-קבוצות, ומסוכמות במקביל.

צריכת הזיכרון מסתכמת ב: $total_memory_size = 4 * 16 * 2^{(8+8)} = 2^{22}\ bit$, הפעם כמות זיכרון קטנה משמעותית מהדוגמה הקודמת אך בכל זאת היא רבה. שיפור בצריכת זיכרון מתקבל ע"י המשך בחלוקת הארגומנטים לתת-קבוצות, לדוגמה, קבוצות של 4 ביט.

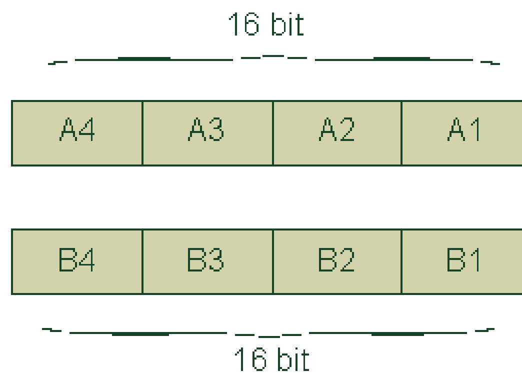

Figure 3

התוצאה תחושב באופן הבא:

$$product = a1*b1 + ((a2*b1) << 4) + ((a3*b1) << 8) + ((a4*b1) << 12) + ((a1*b2 << 4) + ((a2*b2) << 8) + ((a3*b2) << 12) + ((a4*b2) << 16) + ((a1*b3 << 8) + ((a2*b3) << 12) + ((a3*b3) << 16) + ((a4*b3) << 20) + ((a1*b4 << 12) + ((a2*b4) << 16) + ((a3*b4) << 20) + ((a4*b4) << 24)$$

וצריכת הזיכרון מסתכמת ב: $total_memory_size = 16 * 8 * 2^{(4+4)} = 2^{15}\ bit$, הפעם מדובר בצריכת זיכרון סבירה מאוד, אך המכשול הוא מספר רב של דרגות החיבור, אותו יש לבצעה על מנת לקבל את התוצאה הסופית (16 תוצאות ביניים ב-4 דרגות של חיבור). בעבור כל הפתרונות המוצגים מעלה נשתמש בזיכרון ROM המוצג באיור מטה אשר נבנה כקופסה שחורה באמצעות Quartus בדרך הבאה:

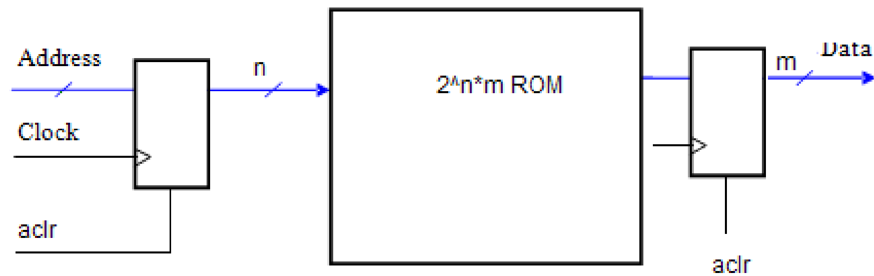

Figure 4

1. Select Tools > MegaWizard Plug-In Manager and the following window appears to ask whether to create a new, edit or copy an existing megafunction.

2. Select Create a new custom megafunction variation and click Next.

3. A new window called page 2a will appear to let you select the type of megafunction you want to design. Select Memory Complier from the tree menu and expand it. Select ROM:1-PORT. Select Cyclone II in Which device family you be using. Select VHDL as the type of output file to create, and give the file the name to be rom.vhd then click Next.

4. In page 3 of the wizard specify the memory size of M x N-bit words, and select M4K as the type of ROM block. Accept the default settings to use a single clock for the ROM's registers then click Next.

5. In page 4, select the setting called 'q' output port under the category "Which ports should be registered"? This setting creates a ROM module that matches the structure in Figure 4, with registered input ports and registered output ports. Select aclear for output register. Click Next.

6. In page 5, the initial content of the memory should be initialized. The memory used in this part is ROM, therefore, the second option of Do you want to specify the memory initial content is enabled. Specify the filename rom.mif in the text box. You will know about the structure of this file at the end of this document. You need this file to specify some data values to be stored in the memory. Click Next.

7. Continue to click Next again and then click Finish in the last page of the wizard.

קובץ אתחול בפורמט MIF יישמר בתיקיית הפרויקט יחד עם ה-ROM שנבנה. את הקובץ rom.vhd יש לצרף לפרויקט הקיים. מיבנה של הקובץ MIF מתואר מטה.

```
DEPTH = 32;
WIDTH = 8;
ADDRESS_RADIX = HEX;  (may be BIN, OCT, DEC, HEX)
DATA_RADIX = BIN;  (may be BIN, OCT, DEC, HEX)
CONTENT
BEGIN

00 : 00000000;
01 : 00000001;
02 : 00000010;
03 : 00000011;
04 : 00000100;
05 : 00000101;
06 : 00000110;
07 : 00000111;
08 : 00001000;
09 : 00001001;
0A : 00001010;
0B : 00001011;
0C : 00001100;
END;
```

בקובץ זה ניתן לייצג data ו-address בבסיסים הבאים: BIN, OCT, DEC, HEX. עמודות השמאליות (שמאלה מנקודתיים) הן עמודות של הכתובת, עמודות מימין לנקודתיים הן העמודות של ה-data.

ניתן להשתמש בשני קבצים mem_init.pdf ו-mem_init_pow.pdf הנמצאים בכתובת הבאה: www.abramovbenjamin.net/labs/ על מנת לחולל קבצי *.mif של זכרון 8*256 ו-16*512 בהתאם.

בניסוי זה נבחן עלויות וביצועיים של שתי גישות שונות למימוש המכפל. גישה אחת היא הגישה המתוארת באיור 3. גישה השנייה שתבחן מתוארת כדלהלן:

$$(a+b)^2-(a-b)^2=4ab;\ then$$

$$ab=\frac{(a+b)^2}{4}-\frac{(a-b)^2}{4}$$

על מנת לחשב כפל של שני ארגומנטים, לדוגמה, של 8 ביט כל אחד, נשתמש בזיכרונות השומרים תוצאות של $\frac{(a+b)^2}{4}$ ושל $\frac{(a-b)^2}{4}$. יתרון של הגישה זאת בכך שבכדי לקבל כפל של ארגומנטים של 8 ביט, יש להשתמש בזיכרון של 512 מילים של 16 ביט מילה לעומת זיכרון של 16*65536 ביט לפי הגישה מאיור 2. מבנה

החומרה המבצעת פעולה זו מוצג באיור 5. ניתן להוכיח ששני הזיכרונות הנמצאים בשימוש בפעולה זו מכילים את אותו התוכן.

במימוש של המשימה זו יש להעביר את כל הכניסות דרך הרגיסטרים, גם התוצאה הסופית יש להעביר דרך הרגיסטר. כל הזיכרונות יש לבנות על פי המודל המוצע- רגיסטר של הכתובת והרגיסטר של המוצא. חשוב לזכור, קוו ה-reset (aclr) במבנים של ALTERA פעיל ברמה לוגית '1' !!!.

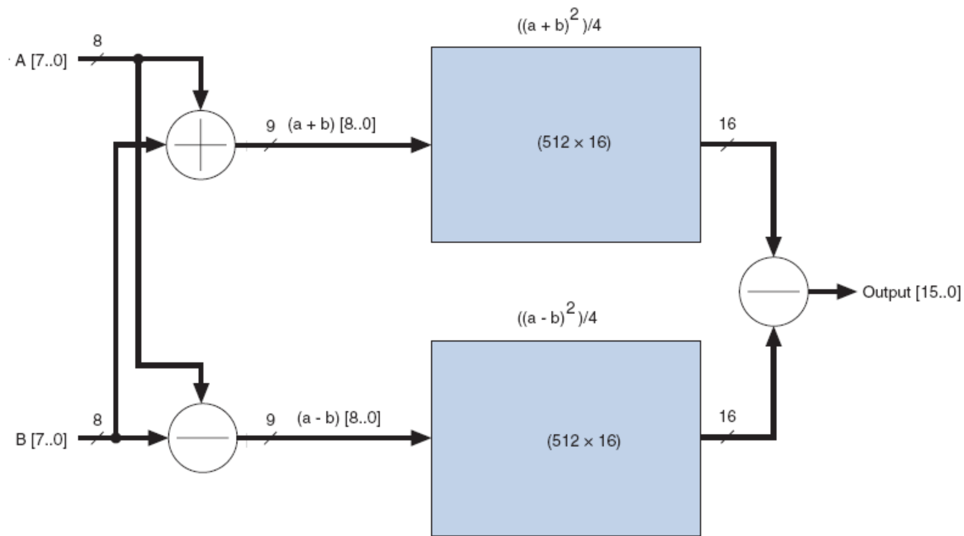

Figure 5

המשימה:

1. כתוב מכפל הכופל שני ארגומנטים של 16 ביט כל אחד בשתי השיטות המוצעות.
2. בצע סימולציה. על מנת להריץ סימולציה יש לבצע שלבים הבאים (בנוסף לקומפילציה של הפרויקט):
   - אחרי מיפוי של ספריית work בצע פקודה – vmap altera_mf work
   - קמפל שני קבצים altera_mf_components.vhd ו-altera_mf.vhd הנמצאים לפי מסלול זה

     C:/altera/10.1/quartus/eda/sim_lib/
   - לאחר מכן יש לקמפל את קבצי הפרויקט על פי הסדר הנדרש
3. תריץ סינתזה ותשווה את הביצועים והעלויות של שני המימושים.