01076009

องค์ประกอบคอมพิวเตอร์และภาษาแอสเซมบลี

Computer Organization and Assembly Language

Arithmetic

Binary Representation

- พื้นฐานของระบบคอมพิวเตอร์คือ เลขฐาน 2

0101 1000 0001 0101 0010 1110 1110 0111

Most significant bit | Least significant bit

- ลำดับเลขฐาน 2 ข้างต้น แทนตัวเลข

$$0 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^{0}$$

- เลขฐาน 2 ขนาด 32 บิต สามารถแทนค่าได้ 2^{32} ตัวเลข (กรณีที่เป็นแบบ unsigned หรือตัวเลขทุกตัวเป็นจำนวนบวก)

Negative Number

- แต่ถ้าเราต้องการแทนค่าให้ได้ทั้งจำนวนบวกและจำนวนลบแล้ว เราก็จะแทนได้เพียงจำนวนบวก 2^{31} ตัว (รวม 0) และจำนวนลบอีก 2^{31} ตัว

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$$
$$\ldots$$
$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$$
$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$$
$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$$
$$\ldots$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$$

2's Complement

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

เหตุใด การแทนเลขจำนวนลบด้วย 2's Complement จึงเป็นที่นิยม?

ลองพิจารณา ผลรวมของ 1 กับ -2 ซึ่งได้ผลลัพธ์เป็น -1

และพิจารณา ผลรวมของ 2 and -1 ซึ่งได้ผลลัพธ์เป็น +1

จะเห็นว่าในรูปแบบนี้ สามารถแสดงผลลัพธ์ของการบวกได้ โดยต้องแปลงอะไรเพิ่มเติมอีก

$$x_{31}\ -2^{31} + x_{30}\ 2^{30} + x_{29}\ 2^{29} + \ldots + x_1\ 2^1 + x_0\ 2^0$$

2's Complement

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

ผลรวมของของจำนวน X ใดๆ กับจำนวน Invert ของ X (x') จะได้เท่ากับตัวเลขฐาน 2 ที่เป็น 1 หมด (-1) เสมอ

$x + x' = -1$

$x' + 1 = -x$ จากสมการนี้ เราสามารถสร้างค่าลบของจำนวนใดๆ

$-x = x' + 1$ โดยการกลับทุกบิตและบวกด้วย 1 ได้

และในทำนองเดียวกัน ผลรวมของ X กับ –X ก็จะได้เท่ากับตัวเลขฐาน 2 ที่เป็น 0 ทั้งหมด โดยมีตัวทด 1

Example

- จงคำนวนเลข 2’s Complement ของเลขฐาน 10 ต่อไปนี้

  5, -5, 6

5:	0000	0000	0000	0000	0000	0000	0000	0101
-5:	1111	1111	1111	1111	1111	1111	1111	1011
-6:	1111	1111	1111	1111	1111	1111	1111	1010

Signed / Unsigned

- จากคำสั่ง ARM ที่กล่าวมาในครั้งก่อน จะเห็นว่าในในระดับ Hardware จะมองข้อมูลจำนวนเต็มออกเป็น 2 แบบ คือ
  - Unsigned (ในภาษาซีจะกำหนดชนิดเป็น unsigned int) โดยตัวเลขจะเป็น บวกทั้งหมด และบิตซ้ายสุดจะเป็นส่วนหนึ่งของค่าข้อมูล
  - Signed (ในภาษาซีจะกำหนดชนิดเป็น signed int หรือ int) โดยตัวเลขสามารถเป็นได้ทั้ง บวกและลบ ซึ่งบิตซ้ายสุดจะทำหน้าที่บอกว่าเป็นจำนวนลบหรือไม่
- ในการประมวลผล เราต้องคำนึงถึงความแตกต่างระหว่างการแทนค่าทั้งสองแบบนี้เสมอ และควรทราบว่าหากไม่มีการใช้ตัวเลขที่เป็นลบ ก็ควรมองแบบ Unsigned เนื่องจากจะสามารถแทนจำนวนเลขได้มากกว่า

Example

- เช่นคำสั่ง

```
        CMP     r0, #0
        BLE     exit
        MOV     r0, #1
exit:
```

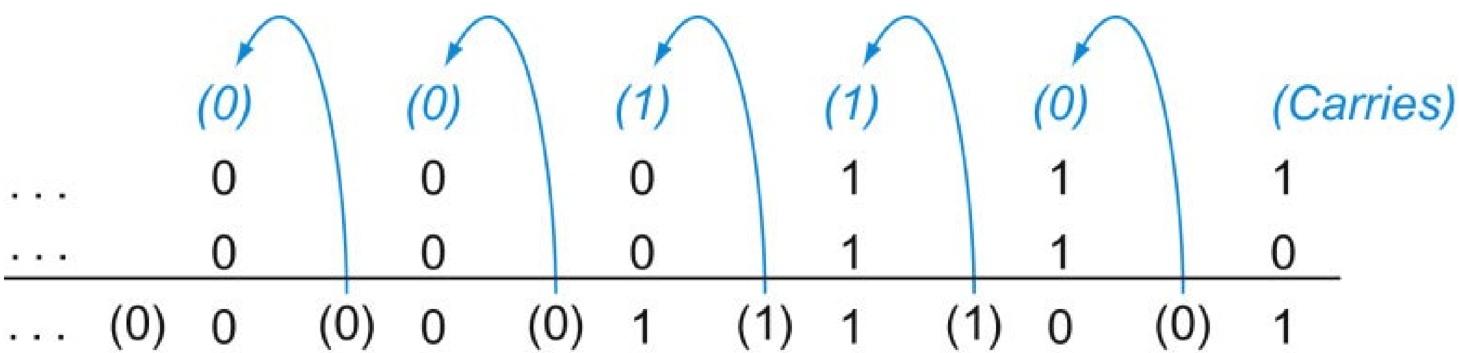
- สมมติให้ r0 มีตัวเลข 1111 01…01
- คำถาม คือ หลังจากการทำงานนี้ r0 จะมีค่าเป็นเท่าไร
- แล้วหากใช้คำสั่ง BLS แทน BLE จะเกิดอะไรขึ้น

Sign Extension

- ข้อมูลตัวเลขที่เก็บในหน่วยความจำ ปกติใน ARM จะใช้ 4 ไบต์ต่อข้อมูล
- แต่ในกรณีที่ข้อมูลมีขนาดเล็ก อาจจะใช้พื้นที่น้อยกว่านั้น เช่น 2 ไบต์ หรือ 1 ไบต์ ในการเก็บ เพื่อประหยัดหน่วยความจำ
- สมมติว่าเก็บข้อมูล 16 บิต ค่าเป็น -1 ก็คือ FF FFh
- แต่เมื่อจะโหลดเข้ามาใน register เพื่อประมวลผล รีจิสเตอร์ของ ARM เป็น 32 บิตทั้งหมด ดังนั้นหากโหลดเข้ามาตรงๆ ก็จะกลายเป็น 00 00 FF FFh ซึ่งทำให้ค่า -1 หายไป
- ดังนั้นจึงต้องมีการทำ sign extension เมื่อมีการโหลดข้อมูลแบบ signed ขนาด 8 หรือ 16 บิต เข้ามา โดยการ copy บิตซ้ายสุดมายังบิตอื่นๆ ด้วย เช่น FF FFh จะเป็น FF FF FF FFh (ARM จะมีคำสั่ง LDRSB และ LDRSH)

Addition and Subtraction

- สำหรับการบวก จะคล้ายกับการบวกเลขฐาน 10 ทั่วไป คือ ถ้ามีการทดก็จะบวกเข้าไปที่หลักหน้า
- สำหรับการลบก็ยังคงใช้การบวก เช่น จาก A –B ก็เปลี่ยนเป็น A + (-B) การสร้าง –B ก็ใช้วิธีการ 2’s Compliment โดยการกลับบิต แล้วบวก 1 เข้าไป



Source: H&P textbook

Overflows

- Overflows คือ เหตุการณ์ที่ขนาดของ register ไม่พอที่จะใส่ขนาดของผลลัพธ์
- สำหรับตัวเลขแบบ Unsigned นั้น Overflow จะเกิดขึ้นเมื่อตัวทดหลักสุดท้ายไม่สามารถรองรับได้ (บางครั้งเรียก Underflow)
- สำหรับการตัวเลขแบบ Signed นั้น Overflow จะเกิดขึ้นเมื่อ MSB (Most Significant Bit) ของผลลัพธ์แตกต่างจาก MSB ของข้อมูลเริ่มต้น
  – เมื่อบวกเลขจำนวนบวก 2 จำนวน แต่ได้ผลลัพธ์เป็นลบ
  – เมื่อบวกเลขจำนวนลบ 2 จำนวน แต่ได้ผลลัพธ์เป็นบวก
  – ผลลัพธ์ของจำนวนบวก กับ จำนวนลบ จะไม่มีทางเกิด Overflow ได้

Overflows

- สมมติว่าเป็นข้อมูลขนาด 8 บิต เพื่อให้เข้าใจง่าย
  - กรณี Unsigned
    - A = 1000 0000, B = 1000 0000
    - ผลลัพธ์จะได้เท่ากับ 1 0000 0000 ซึ่ง Overflow
  - กรณี Signed
    - A = 0100 0000, B = 0100 0000
    - ผลลัพธ์จะได้เท่ากับ 1000 0000 ซึ่งจะเห็นว่าทั้งคู่เป็นเลขบวก แต่ผลลัพธ์เป็นลบ
    - A = 1000 0001, B = 1000 0001
    - ผลลัพธ์จะได้เท่ากับ 1 0000 0010 ซึ่งเป็นเลขลบ (ตัดตัวทด) แต่ผลลัพธ์จะเป็นบวก
    - A = 1111 1111 (-1), B = 0000 0001 (1)
    - ผลลัพธ์จะได้ 1 0000 0000 ซึ่งเมื่อตัดตัวทดไป จะได้ค่า 0 ซึ่งไม่ Overflow

Exercise

- กำหนดให้ตัวเลขแบบ Unsigned 8 บิต 2 จำนวน ให้หาว่า 69 – 90 เกิด Overflow หรือ Underflow หรือไม่

  0100 0101 - 0101 1010 = 1110 1011 -> Underflow

- กำหนดให้ตัวเลขแบบ Unsigned 8 บิต 2 จำนวน ให้หาว่า 102 – 44 เกิด Overflow หรือ Underflow หรือไม่

  01100110 - 00101100 = 00111010 -> ไม่ Error

Exercise

- กำหนดให้ตัวเลขแบบ Signed 8 บิต 2 จำนวน ให้หาว่า 200 + 103 เกิด Overflow หรือ Underflow หรือไม่

  200 = -56 (1100 1000)

  1100 1000 + 0110 0111 = 1 0010 1111 (=47_{10}) -> ไม่ Overflow

- กำหนดให้ตัวเลขแบบ Signed 8 บิต 2 จำนวน ให้หาว่า 247 + 237 เกิด Overflow หรือ Underflow หรือไม่

  247 = -9 (11110111), 237 = -19 (11101101)

  11110111 + 11101101 = 1 1110 0100 (=-28_{10}) -> ไม่ Overflow

Multiplication

Multiplicand	1000_{ten}
Multiplier	x 1001_{ten}

	1000
	0000
	0000
	1000

Product	1001000_{ten}

การทำงานในแต่ละ Step

1. พิจารณาบิตสุดท้ายของตัวคูณ (Multiplier)
2. ถ้าบิตมีค่าเป็น 1 ให้บวกตัวตั้ง (Multiplicand) เข้ากับผลลัพธ์ (Product)
3. Shift ตัวตั้งไป 1 หลัก (ทางซ้าย)
4. Shift ตัวคูณไป 1 หลัก (ทางขวา)
5. กลับไปทำขั้นตอนที่ 1 จนกว่าจะหมด

Multiplication

- 1010 x 1001
  - บิตสุดท้าย = 1 -> บวกตัวตั้งเข้ากับผลลัพธ์ = 1010
  - Shift Left ตัวตั้งไป 1 หลัก = 10100
  - Shift Right ตัวคูณไป 1 หลัก = 100
  - บิตสุดท้าย = 0 ไม่บวก
  - Shift Left ตัวตั้งไป 1 หลัก = 101000
  - Shift Right ตัวคูณไป 1 หลัก = 10
  - บิตสุดท้าย = 0 ไม่บวก
  - Shift Left ตัวตั้งไป 1 หลัก = 1010000
  - Shift Right ตัวคูณไป 1 หลัก = 1
  - บิตสุดท้าย = 1 -> บวกตัวตั้งเข้ากับผลลัพธ์ = 1011010 เป็นคำตอบสุดท้าย

Multiplication

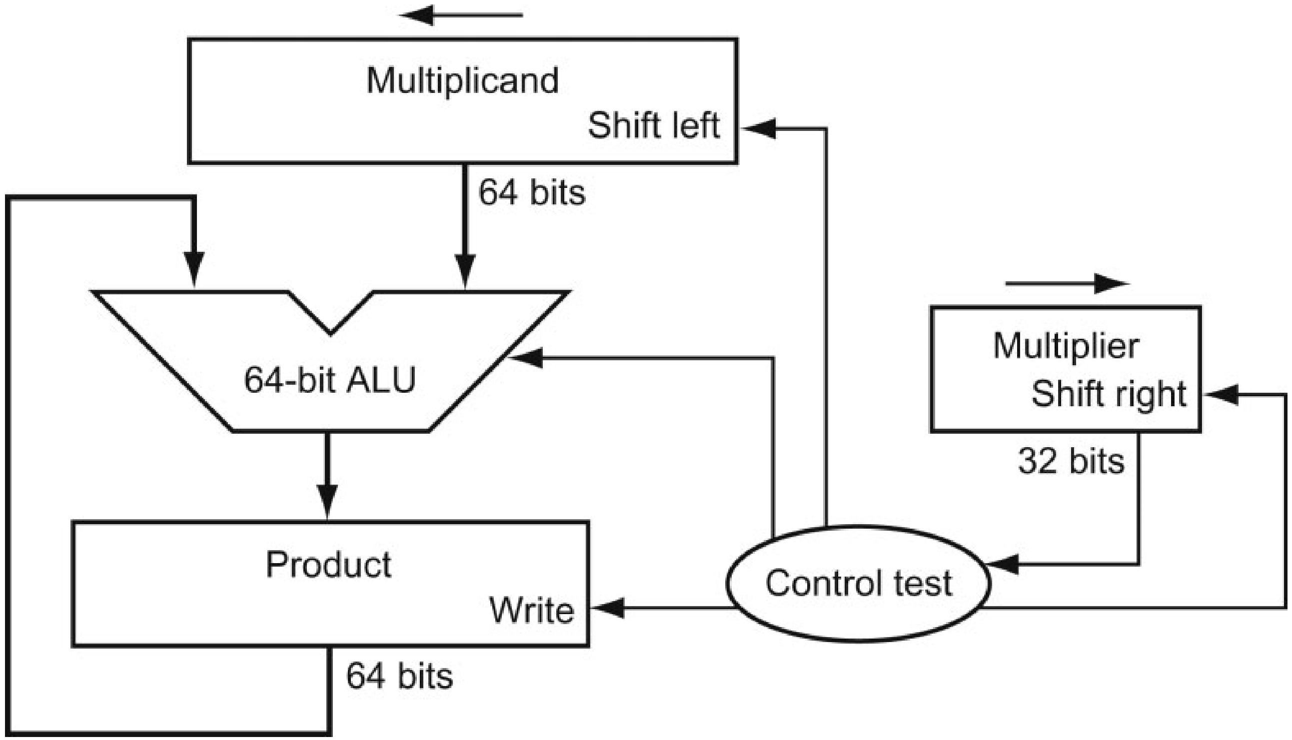
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 $\Longrightarrow$ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 $\Longrightarrow$ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 $\Longrightarrow$ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 $\Longrightarrow$ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Exercise

- ตามตัวอย่างตารางใน Slide ก่อนหน้านี้ ให้แสดงการคูณของ 44_{10} กับ 55_{10}

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	110111	0010 1100	0000 0000
1	1a: 1 -> Prod = Prod + Mcand	110111	0010 1100	0010 1100
	2: Shift Left Multiplicand	110111	0101 1000	0010 1100
	3: Shift Right Multiplier	011011	0101 1000	0010 1100
2	1a: 1 -> Prod = Prod + Mcand	011011	0101 1000	1000 0100
	2: Shift Left Multiplicand	011011	1011 0000	1000 0100
	3: Shift Right Multiplier	001101	1011 0000	1000 0100
3	1a: 1 -> Prod = Prod + Mcand	001101	1011 0000	1 0011 0100
	2: Shift Left Multiplicand	001101	1 0110 0000	1 0011 0100
	3: Shift Right Multiplier	000110	1 0110 0000	1 0011 0100
4	1: 0 -> No Operation	000110	1 0110 0000	1 0011 0100
	2: Shift Left Multiplicand	000110	10 1100 0000	1 0011 0100
	3: Shift Right Multiplier	000011	10 1100 0000	1 0011 0100
5	1a: 1 -> Prod = Prod + Mcand	000011	10 1100 0000	11 1111 0100
	2: Shift Left Multiplicand	000011	101 1000 0000	11 1111 0100
	3: Shift Right Multiplier	000001	101 1000 0000	11 1111 0100
6	1a: 1 -> Prod = Prod + Mcand	000001	101 1000 0000	1001 0111 0100
	2: Shift Left Multiplicand	000001	1011 0000 0000	1001 0111 0100
	3: Shift Right Multiplier	000000	1011 0000 0000	1001 0111 0100

Hardware Algorithm 1

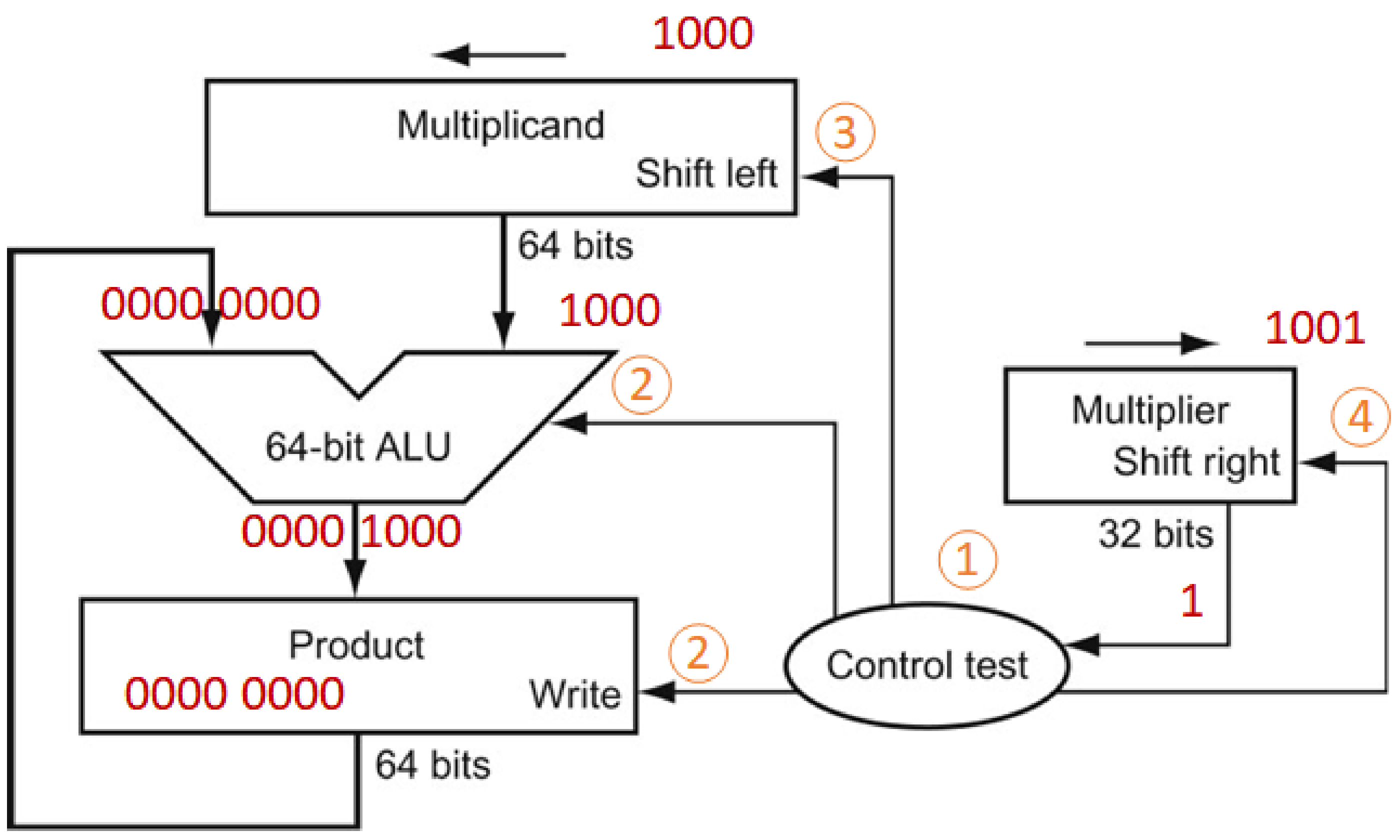


การทำงานในแต่ละ Step

1. พิจารณาบิตสุดท้ายของตัวคูณ (Multiplier)
2. ถ้าบิตมีค่าเป็น 1 ให้บวกตัวตั้ง (Multiplicand) เข้ากับผลลัพธ์ (Product)
3. Shift ตัวตั้งไป 1 หลัก (ทางซ้าย) และ Shift ตัวคูณไป 1 หลัก (ทางขวา)
4. กลับไปทำขั้นตอนที่ 1 จนกว่าจะหมด

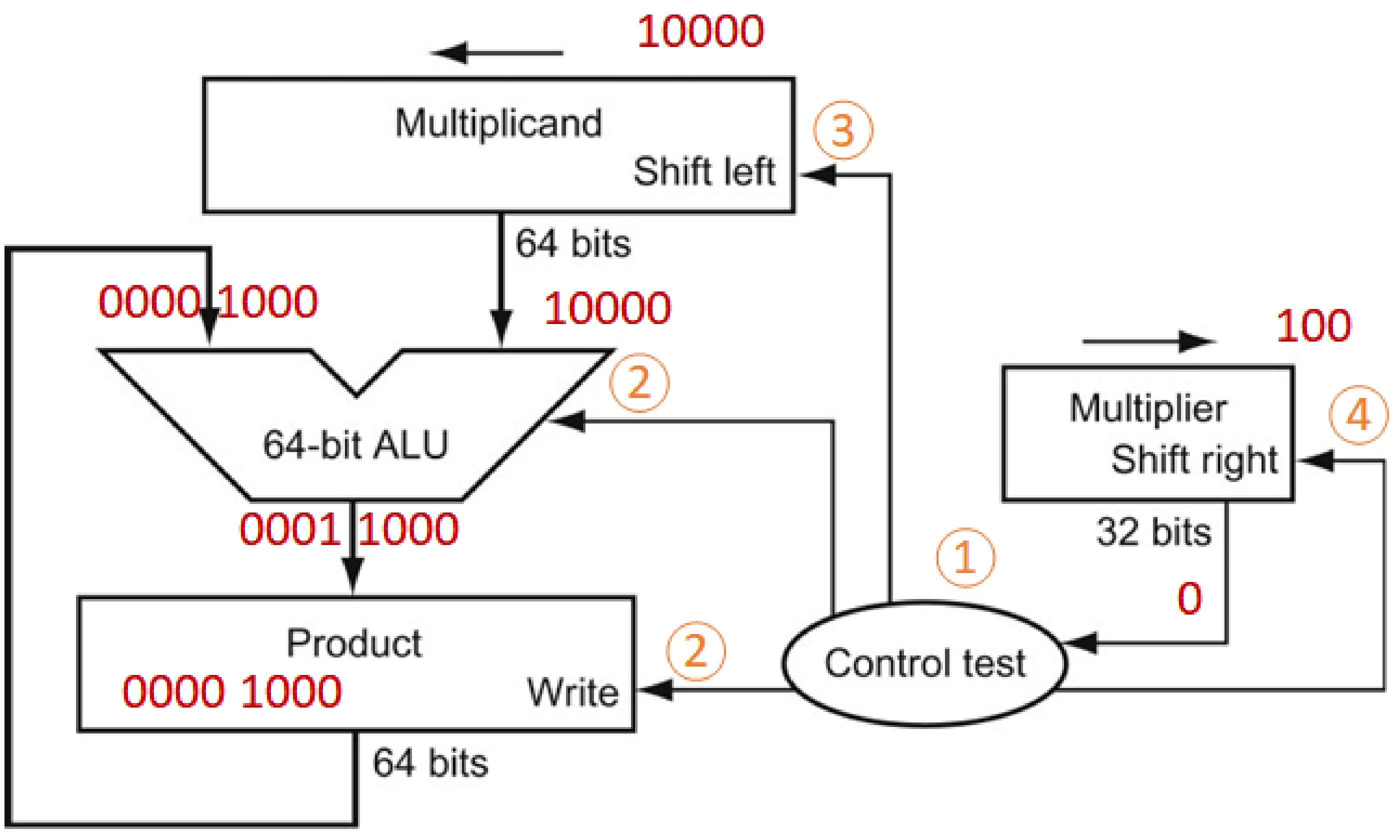
Hardware Algorithm 1

- รอบที่ 1



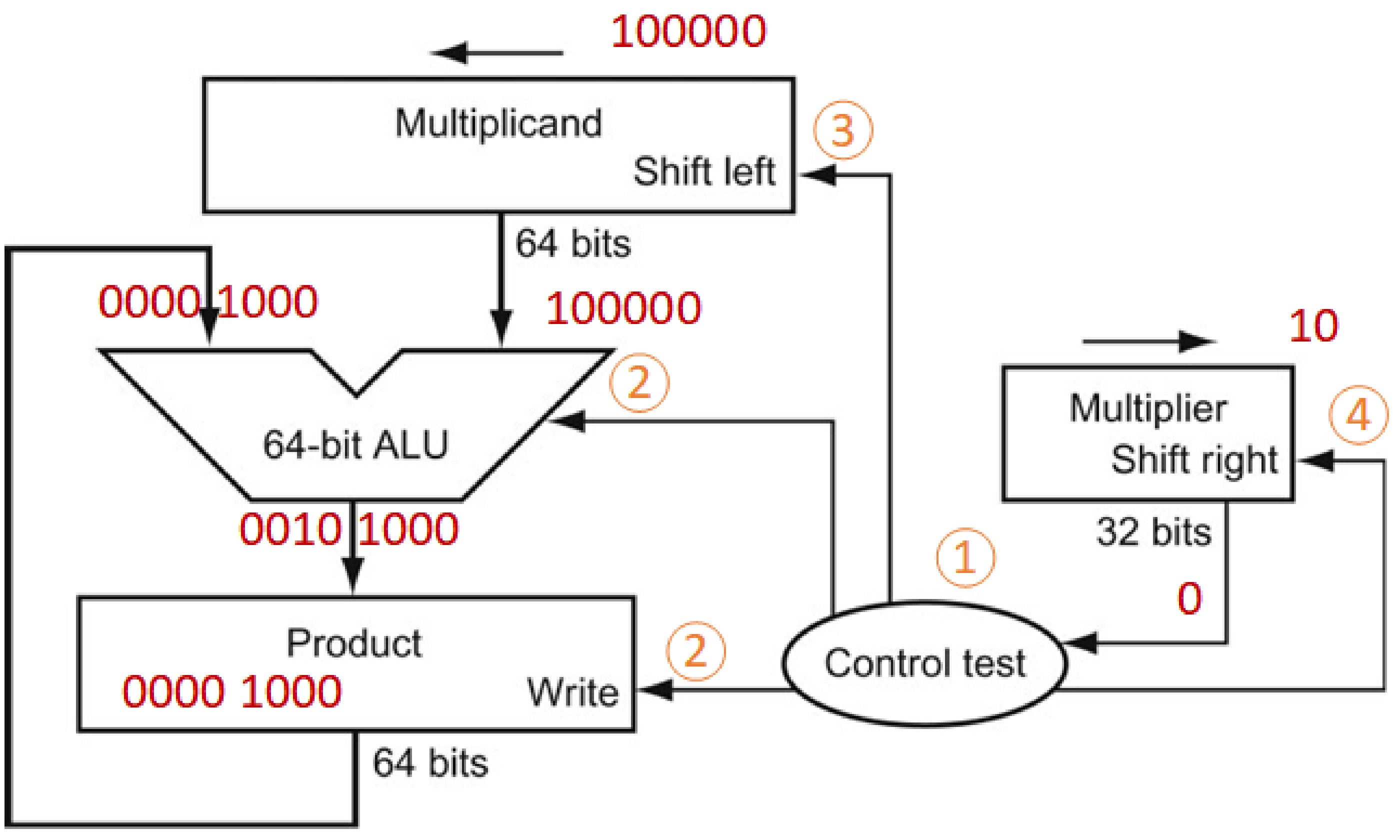
Hardware Algorithm 1

- รอบที่ 2



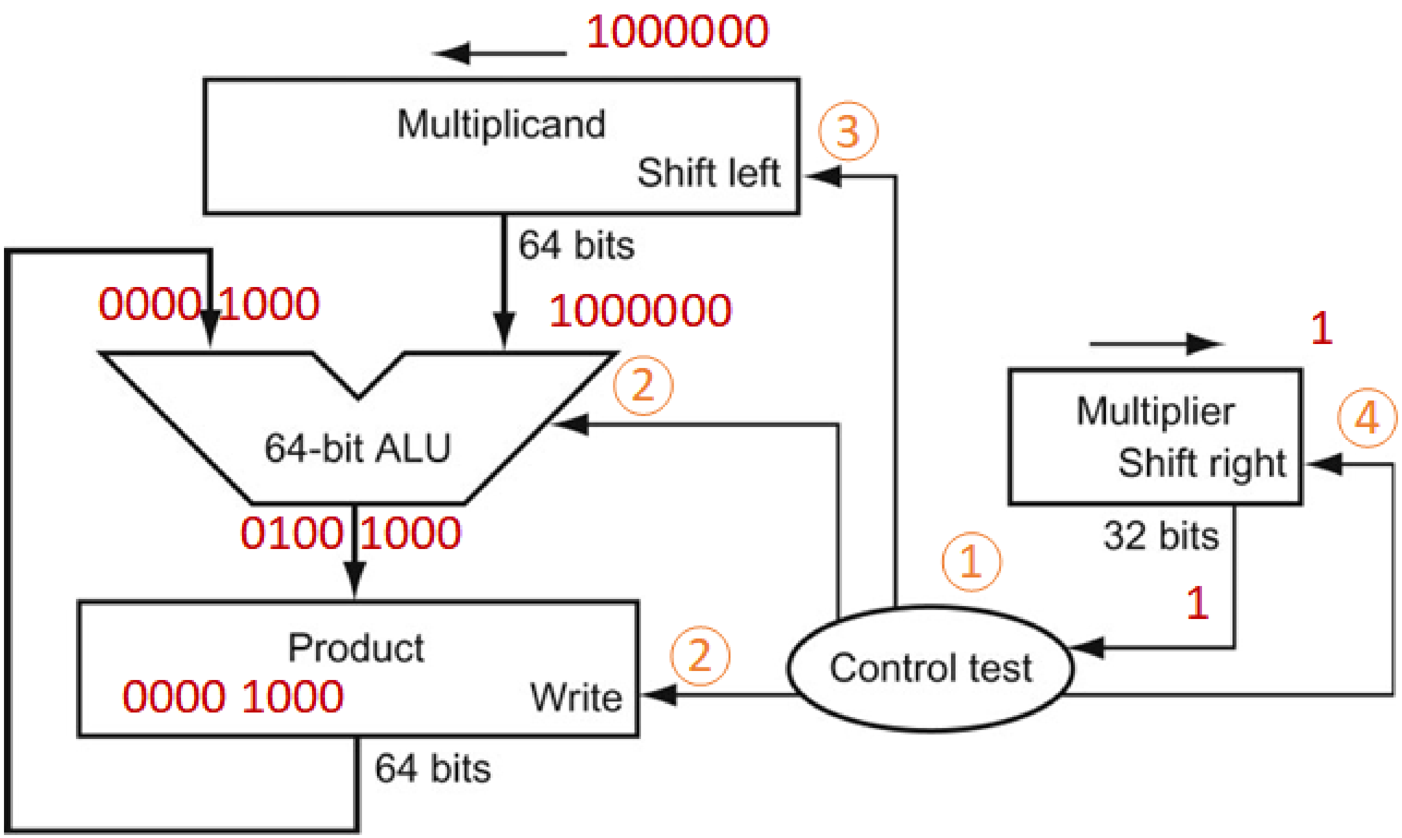
Hardware Algorithm 1

- รอบที่ 3



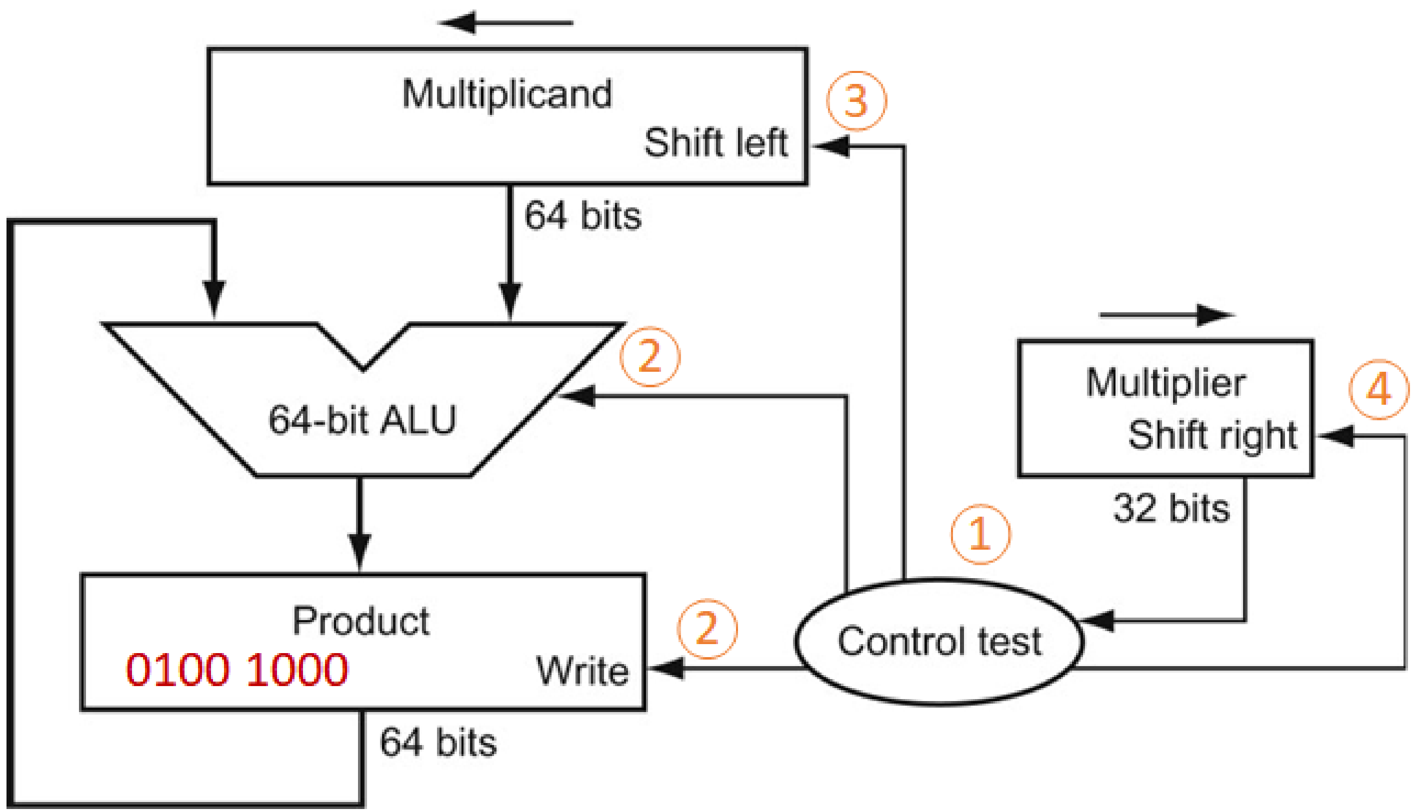
Hardware Algorithm 1

- รอบที่ 4



Hardware Algorithm 1

- Final



Exercise

- ให้แสดงการคูณของ 44_{10} กับ 55_{10} โดยแสดงข้อมูลใน register ในแต่ละ step

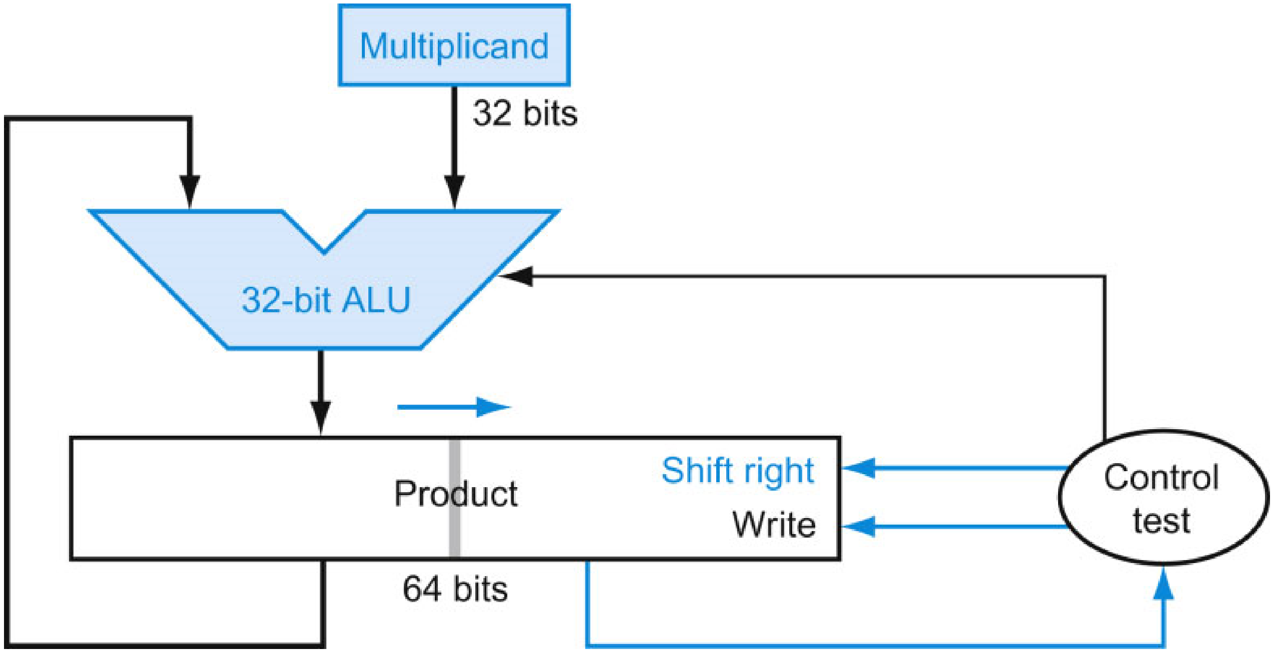
Iteration	Multiplier	Multiplicand	Product
1	110111	0010 1100	0000 0000
2	011011	0101 1000	1000 0100
3	001101	1011 0000	1 0011 0100
4	000110	1 0110 0000	1 0011 0100
5	000011	10 1100 0000	11 1111 0100
6	000001	101 1000 0000	1001 0111 0100

Hardware Algorithm 1

- สมมติว่าใน Hardware ตามรูป ใช้ Clock กำกับการทำงาน โดย 1 Clock เป็นการสั่งให้ทำงาน 1 การทำงาน หากต้องการคูณตัวเลข 32 บิต จะใช้กี่ Clock ในการทำงาน
- มีขั้นตอนใดที่สามารถรวมเข้าด้วยกันได้บ้าง
- หลังจากรวมแล้ว เหลือกี่ Clock ต่อการคูณตัวเลข 32 บิต 2 จำนวน

Hardware Algorithm 2

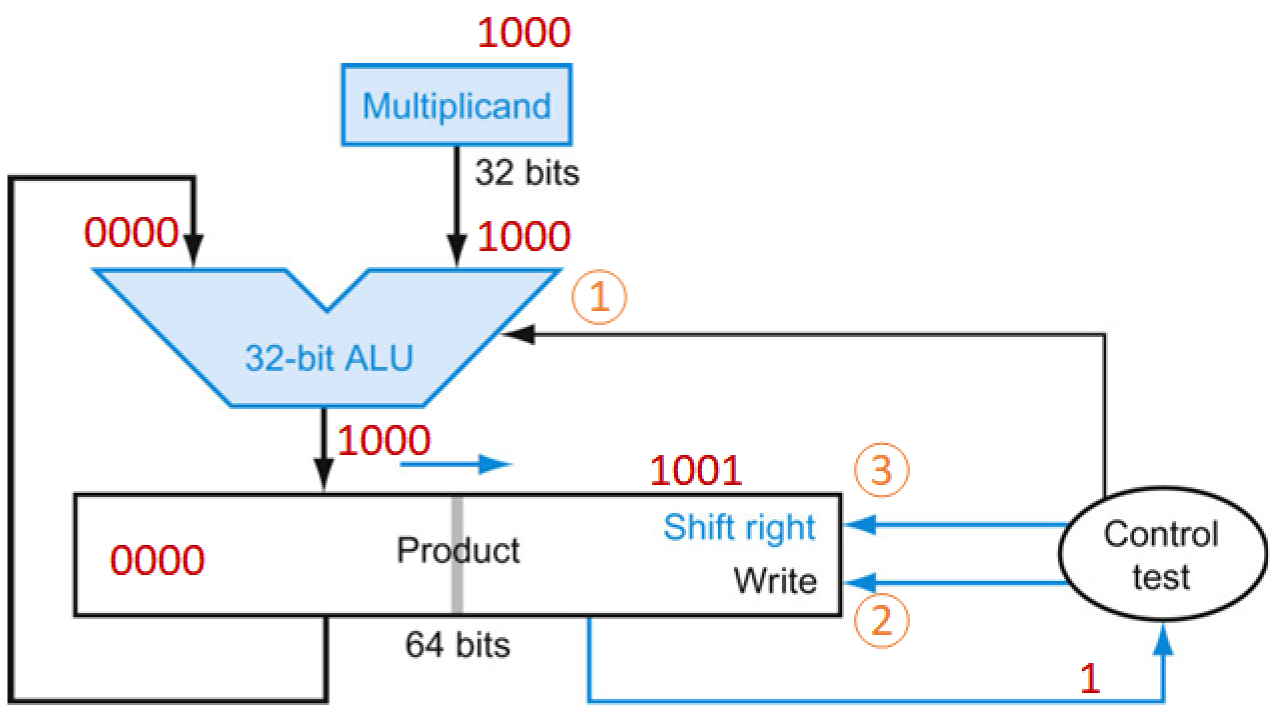
- เพื่อให้สามารถทำงานได้เร็วขึ้น จึงมีการปรับปรุง Hardware ตามรูป



- ALU กับ ตัวตั้งยังเหมือนเดิม มีการรวมตัวคูณกับผลลัพธ์เข้าด้วยกัน
- ในแต่ละ step Product + Multiplier = 64 บิต โดย Product จะเพิ่มขึ้นเรื่อยๆ ครั้งละ 1 บิต แต่ Multiplier จะลดลงเรื่อยๆ ครั้งละ 1 บิต สุดท้าย Product = 64 บิต

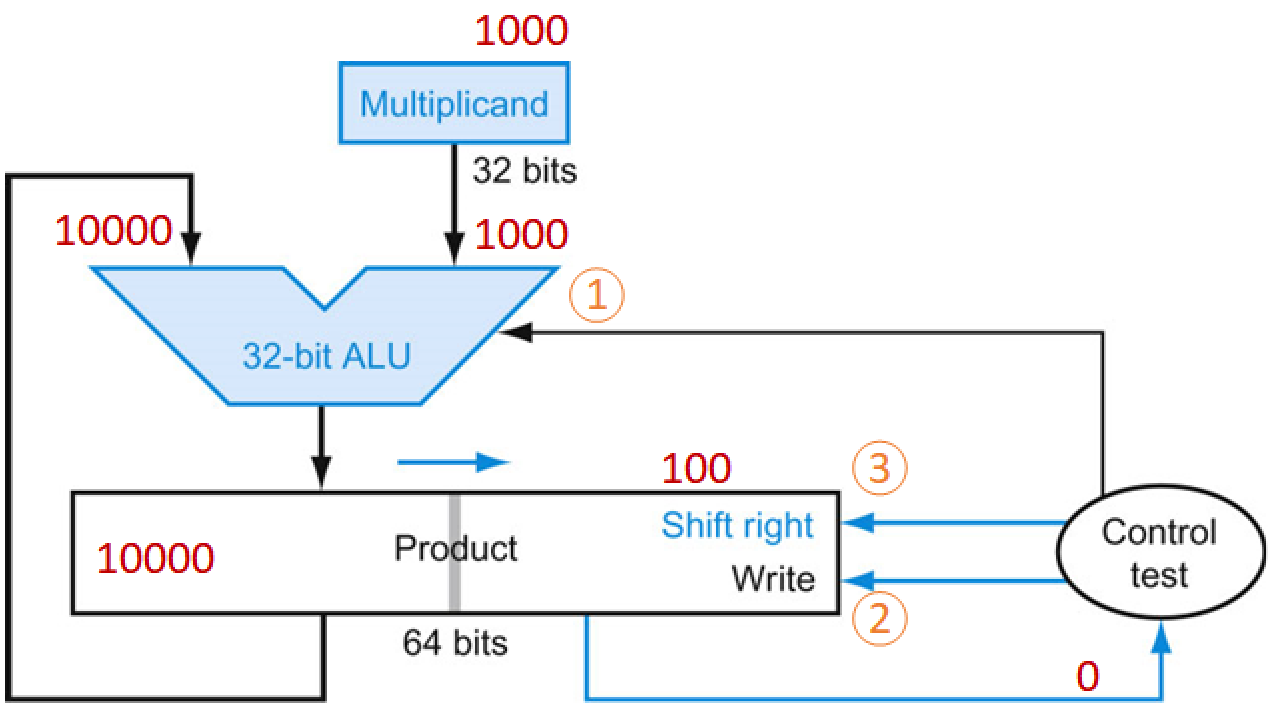
Hardware Algorithm 2

- รอบที่ 1 บิตขวาสุดของตัวคูณ คือ 1 ดังนั้นจะสั่งให้บวก และ write ผลลัพธ์ลงใน Product (ในฝั่ง 32 บิต) จากนั้นสั่ง shift



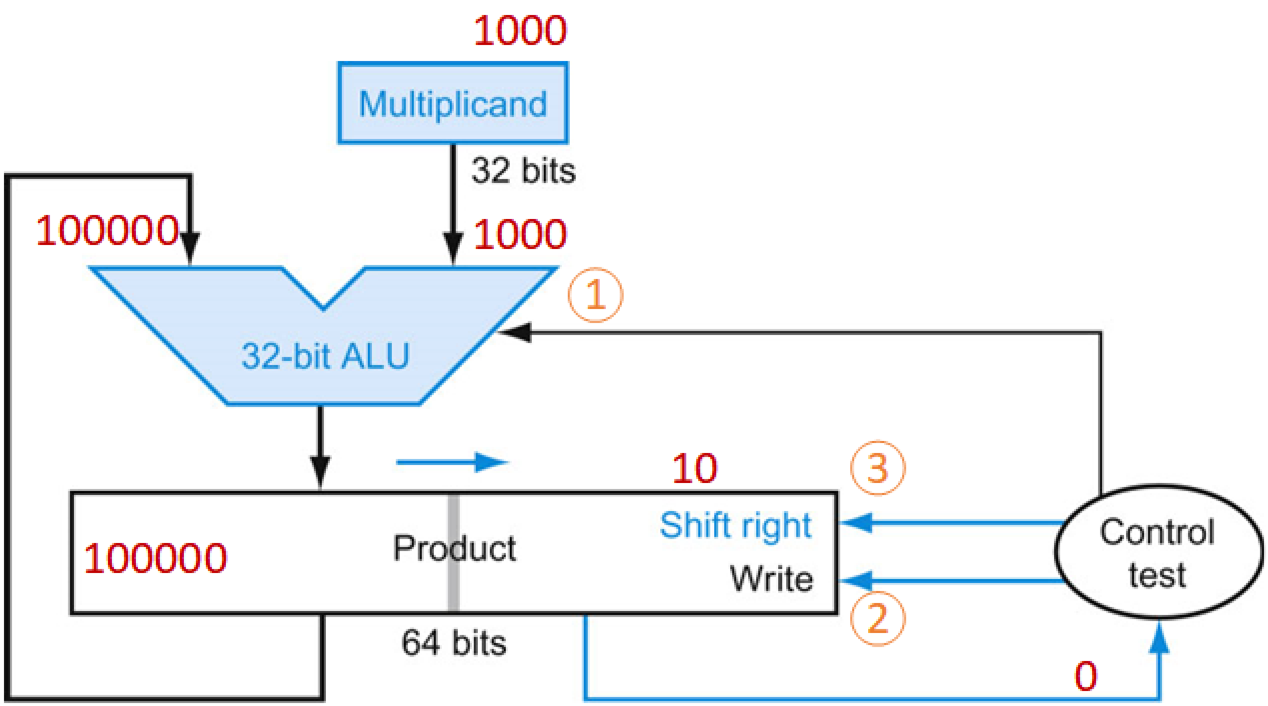
Hardware Algorithm 2

- รอบที่ 2 บิตขวาสุดของตัวคูณ คือ 0 จะไม่บวกและ write ผลลัพธ์ลงใน Product (ในฝั่ง 32 บิต) จากนั้นสั่ง shift



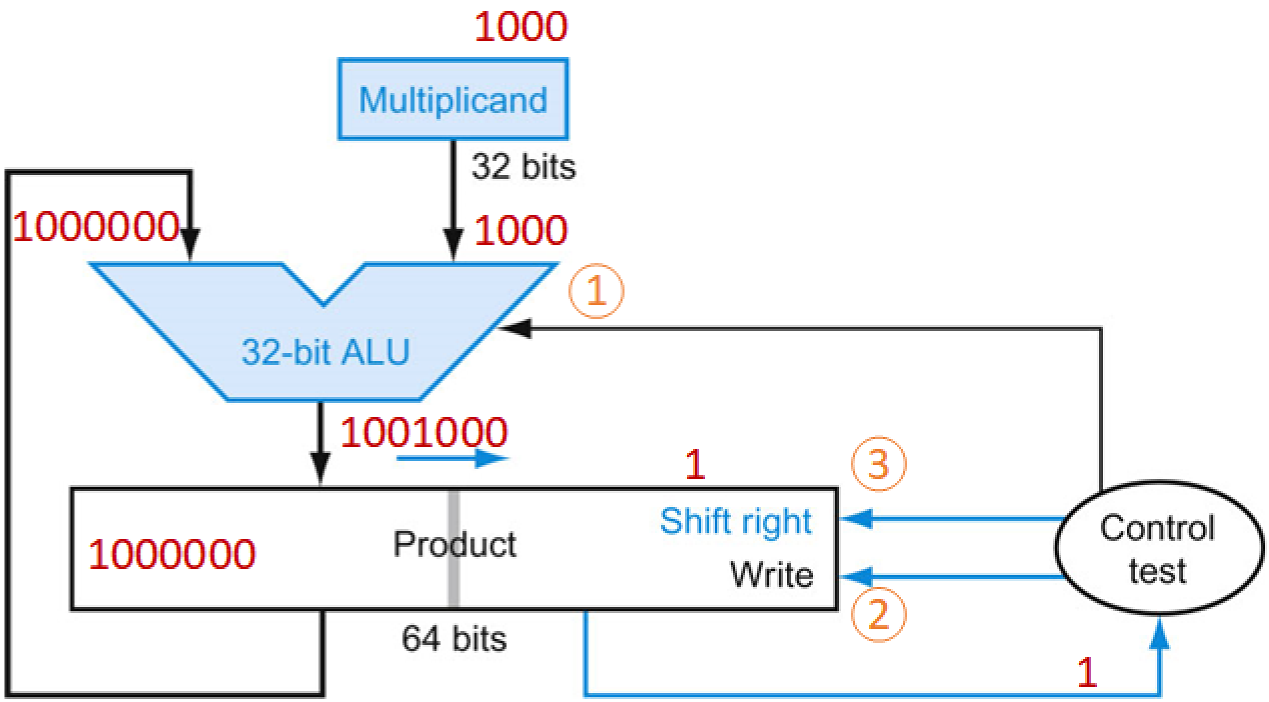
Hardware Algorithm 2

- รอบที่ 3 บิตขวาสุดของตัวคูณ คือ 0 จะไม่บวกและ write ผลลัพธ์ลงใน Product (ในฝั่ง 32 บิต) จากนั้นสั่ง shift



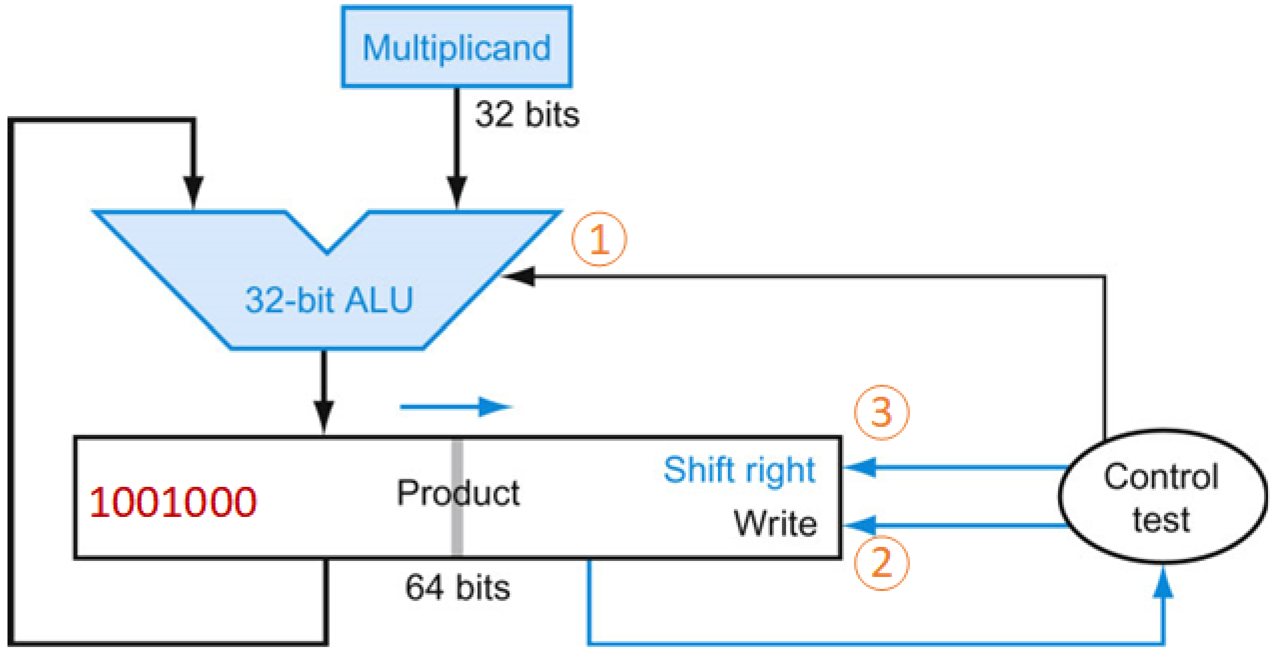
Hardware Algorithm 2

- รอบที่ 4 บิตขวาสุดของตัวคูณ คือ 1 ดังนั้นจะสั่งให้บวก และ write ผลลัพธ์ลงใน Product (ในฝั่ง 32 บิต) จากนั้นสั่ง shift



Hardware Algorithm 2

- ผลลัพธ์สุดท้าย



Exercise

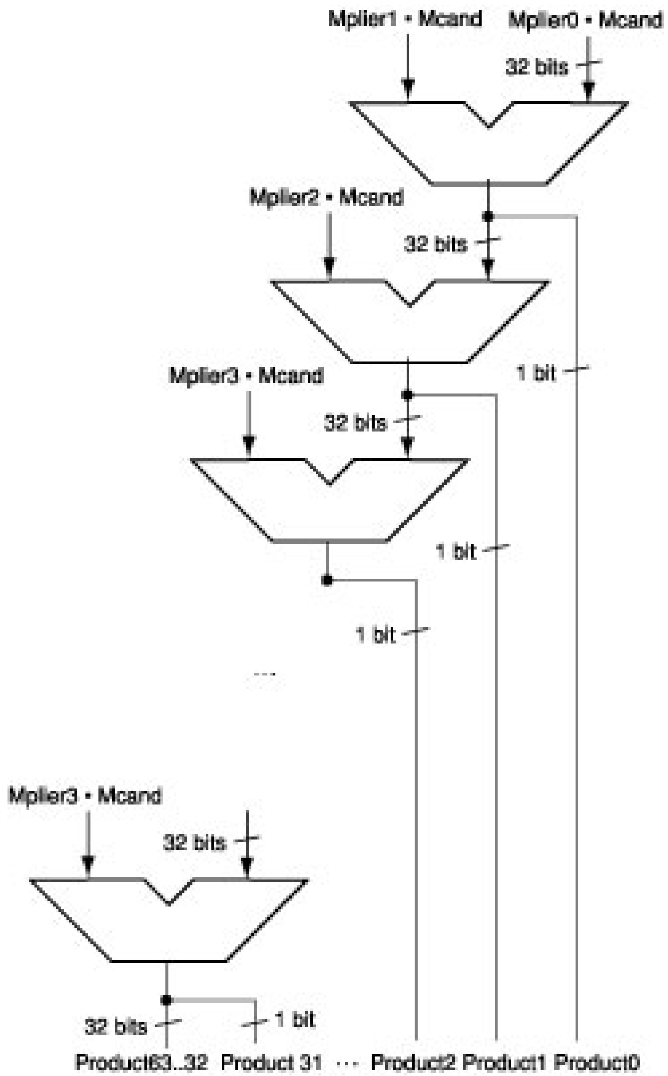
- ให้แสดงการคูณของ 44_{10} กับ 55_{10} โดยแสดงข้อมูลใน register ในแต่ละ step

Iteration	Multiplicand	Product
1	0010 1100	0000 0000 0011 0111
2	0010 1100	1000 0100 0001 1011
3	0010 1100	1001 1010 0000 1101
4	0010 1100	1001 1010 0000 0110
5	0010 1100	1111 1101 0000 0011
6	0010 1100	1001 0111 0100 0001
7	0010 1100	1001 0111 0100 0000

Hardware Algorithm 2

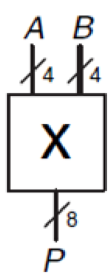
- Hardware ที่สร้างใหม่ มีการทำงานที่เร็วขึ้น มีวงจรน้อยลง
- นอกจากนั้นยังสามารถทำงานกับตัวเลขแบบ Signed ได้ (ในรูปแบบ 2’s Compliment)
- อาจจะเปลี่ยนจากลบเป็นบวก แล้วเมื่อคูณแล้วค่อยเอาเครื่องหมายใส่กลับเข้าไป
- โดยทั่วไป ผลลัพธ์ของการคูณ 32 บิต จะไม่เกิน 64 บิต แต่สำหรับ ARM จะมี 2 คำสั่ง
  - MUL เมื่อคูณแล้วผลลัพธ์ต้องไม่เกิน 32 บิต
  - MULL เมื่อคูณแล้วผลลัพธ์จะเป็น 64 บิต ดังนั้นต้องใช้รีจิสเตอร์ในการเก็บ 2 ตัว เช่น UMULL R1,R4,R2,R3 ; R4,R1:=R2*R3

Fast Multiplication

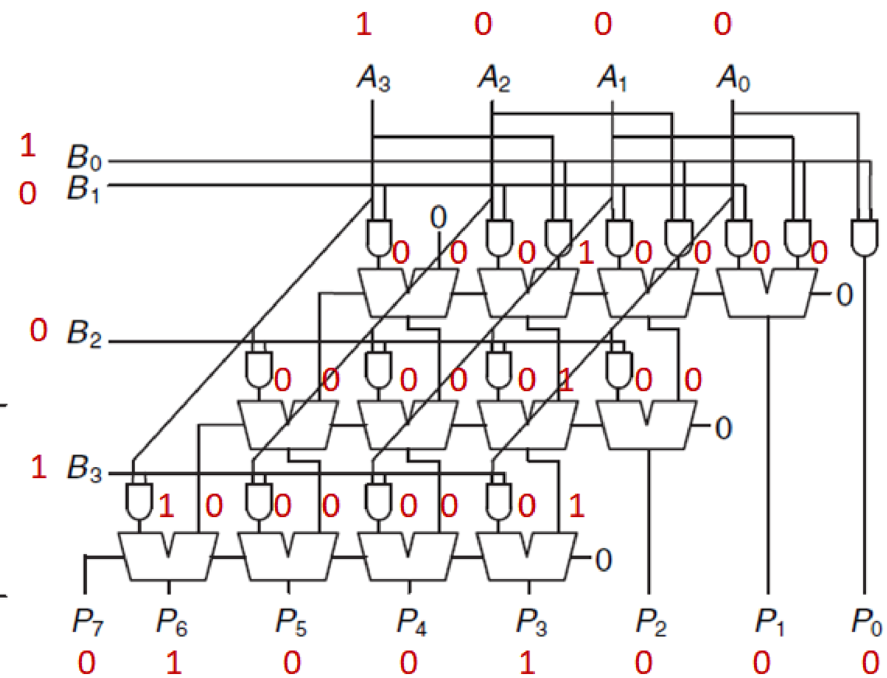


- ใน Hardware ก่อนหน้านี้ จำเป็นต้องมีสัญญาณ clock กำกับ เพื่อให้แน่ใจได้ว่าการบวกจะเกิดขึ้นก่อน shift
- แต่เนื่องจากทำทีละบิต ดังนั้นจึงเลี่ยงไม่ได้ที่จะต้องใช้ clock จำนวนมากในการทำงาน
- สำหรับ Hardware แบบ fast multiplication นี้ จะทำทุกบิตพร้อมๆ กัน ทำให้ลดจำนวน clock ไปมาก
- แต่ก็แลกมาด้วย จำนวน transistor

Fast Multiplication

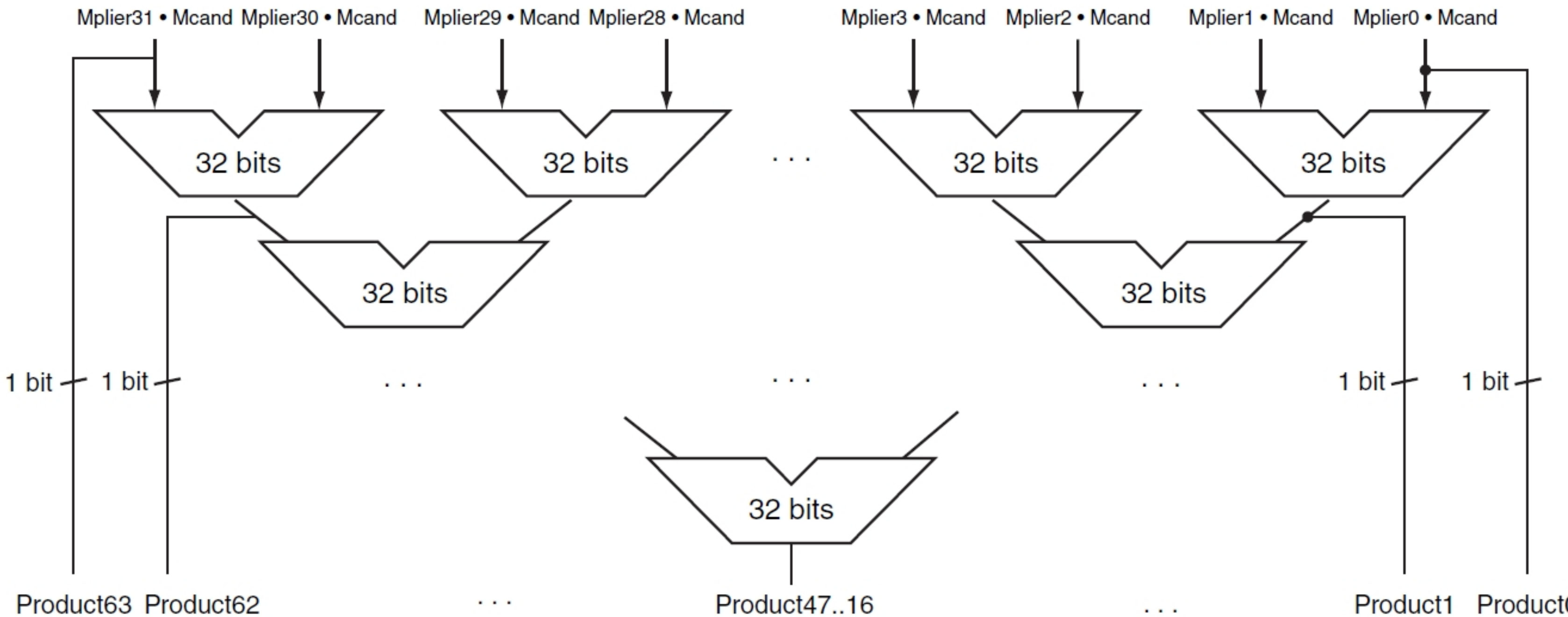


					1	0	0	0
					1	0	0	1
					A_3	A_2	A_1	A_0
×					B_3	B_2	B_1	B_0
					A_3B_0	A_2B_0	A_1B_0	A_0B_0
				A_3B_1	A_2B_1	A_1B_1	A_0B_1	
			A_3B_2	A_2B_2	A_1B_2	A_0B_2		
+		A_3B_3	A_2B_3	A_1B_3	A_0B_3			
	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0



Fast Multiplication

- จาก Hardware ข้างต้น (s.27) จะเห็นว่าจะต้องมี Adder Module ถึง 32 หน่วย
- แต่เราสามารถลดจำนวนหน่วยได้ โดยจัดเรียงการทำงานเสียใหม่ ดังนี้



Division

Quotient: $\overline{\quad 1001_{ten}}$

Divisor: 1000_{ten} | 1001010_{ten} Dividend

$$
\begin{array}{r}
\underline{-1000} \\
10 \\
101 \\
1010 \\
\underline{-1000} \\
10_{ten}
\end{array}
$$

Remainder

- การทำงานในแต่ละ Step
  - Shift ตัวหารทางขวา และเปรียบเทียบกับตัวตั้ง
  - ถ้าตัวหารยังคงมีค่ามากกว่า ให้ shift 0 เข้าไปที่ผลลัพธ์
  - ถ้าตัวหารน้อยกว่า ให้ทำการลบ แล้วสร้างตัวตั้งตัวใหม่ และ shift 1 เข้าไปที่ผลลัพธ์

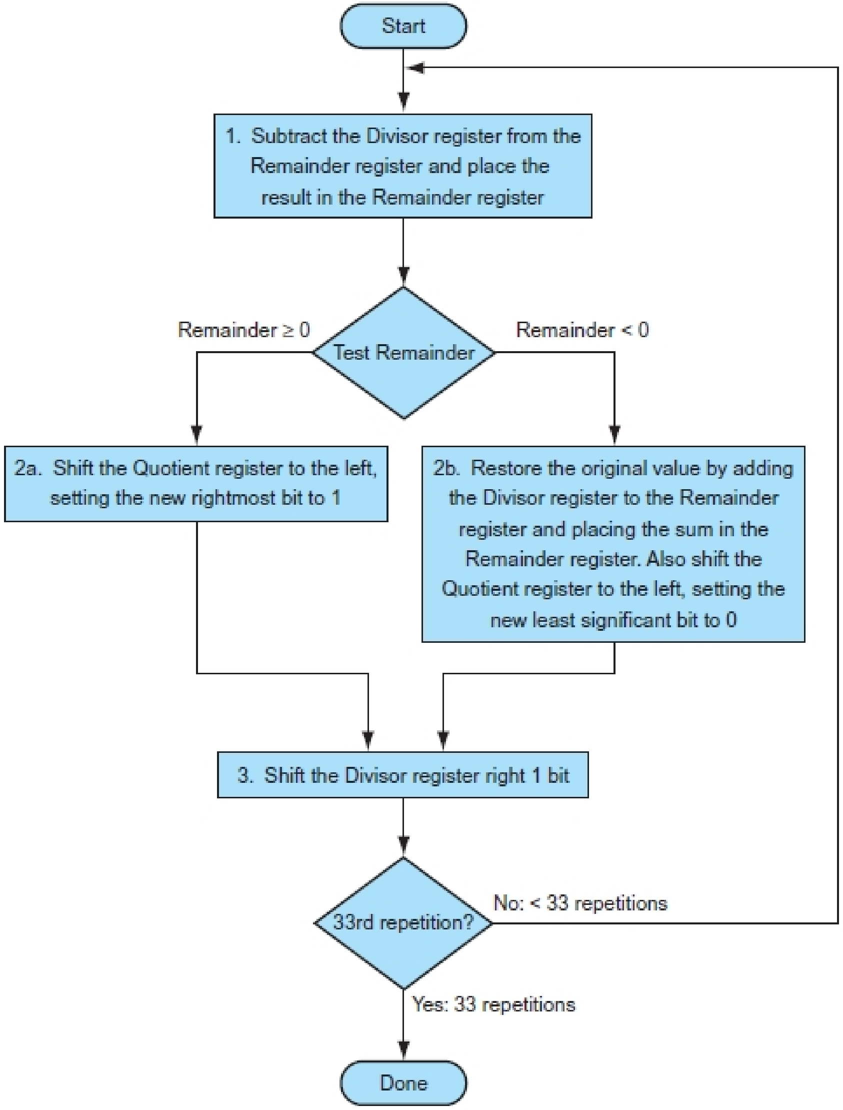
Division

		1001	Quotient
Divisor	1000	1001010	Dividend

	01001010		01001010		00001010		00001010
	100000000000	→	01000000	→	00100000	→	00001000
Quo:	0		000001		000010		00001001

- การทำงานในแต่ละ Step
  - Shift ตัวหารทางขวา และเปรียบเทียบกับตัวตั้ง
  - ถ้าตัวหารยังคงมีค่ามากกว่า ให้ shift 0 เข้าไปที่ผลลัพธ์
  - ถ้าตัวหารน้อยกว่า ให้ทำการลบ แล้วสร้างตัวตั้งตัวใหม่ และ shift 1 เข้าไปที่ผลลัพธ์

Division


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

Divide Example

- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 ➔ +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 ➔ shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001

Exercise

- ให้แสดงการหาร 40_{10} ด้วย 19_{10} ตามตัวอย่างใน slide ก่อนหน้า

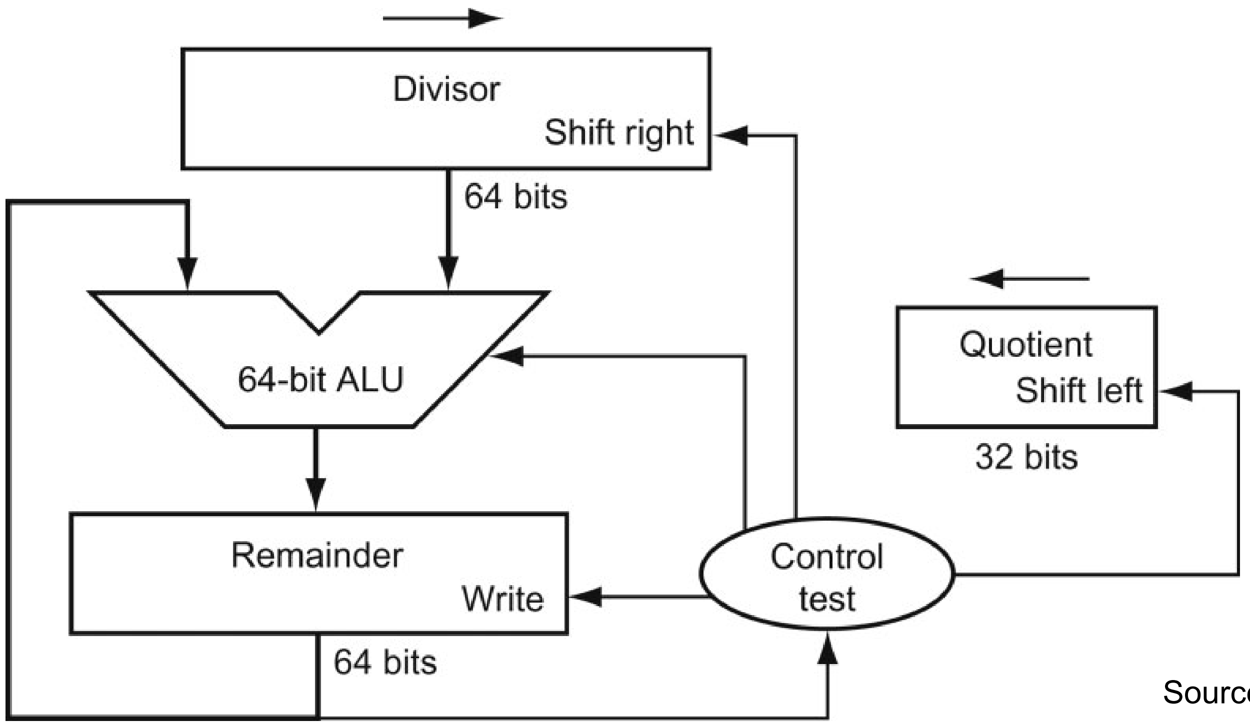
Iteration	Step	Quotient	Divisor	Reminder
0	Initial Values		0001 0011 0000 0000	0010 1000
1	Reminder = Reminder - Divisor		0001 0011 0000 0000	< 0
	Reminder < 0 -> Shift 0 into Q	0		
	Shift Divisor Right		0000 1001 1000 0000	0010 1000
2	Reminder = Reminder - Divisor		0000 1001 1000 0000	< 0
	Reminder < 0 -> Shift 0 into Q	00		
	Shift Divisor Right		0000 0100 1100 0000	0010 1000
3	Reminder = Reminder - Divisor		0000 0100 1100 0000	< 0
	Reminder < 0 -> Shift 0 into Q	000		
	Shift Divisor Right		0000 0010 0110 0000	0010 1000
4	Reminder = Reminder - Divisor		0000 0010 0110 0000	< 0
	Reminder < 0 -> Shift 0 into Q	0000		
	Shift Divisor Right		0000 0001 0011 0000	0010 1000

Exercise

- ให้แสดงการหาร 40_{10} ด้วย 19_{10} ตามตัวอย่างใน slide ก่อนหน้า

Iteration	Step	Quotient	Divisor	Reminder
5	Reminder = Reminder - Divisor		0000 0001 0011 0000	< 0
	Reminder < 0 -> Shift 0 into Q	00000		
	Shift Divisor Right		0000 0000 1001 1000	0010 1000
6	Reminder = Reminder - Divisor		0000 0000 1001 1000	< 0
	Reminder < 0 -> Shift 0 into Q	000000		
	Shift Divisor Right		0000 0000 0100 1100	0010 1000
7	Reminder = Reminder - Divisor		0000 0000 0100 1100	< 0
	Reminder < 0 -> Shift 0 into Q	0000000		
	Shift Divisor Right		0000 0000 0010 0110	0010 1000
8	Reminder = Reminder - Divisor		0000 0000 0010 0110	10
	Reminder > 0 -> Shift 1 into Q	00000001		
	Shift Divisor Right		0000 0000 0001 0011	10
9	Reminder = Reminder - Divisor		0000 0000 0001 0011	< 0
	Reminder < 0 -> Shift 0 into Q	000000010		
	Shift Divisor Right		0000 0000 0000 0100	10

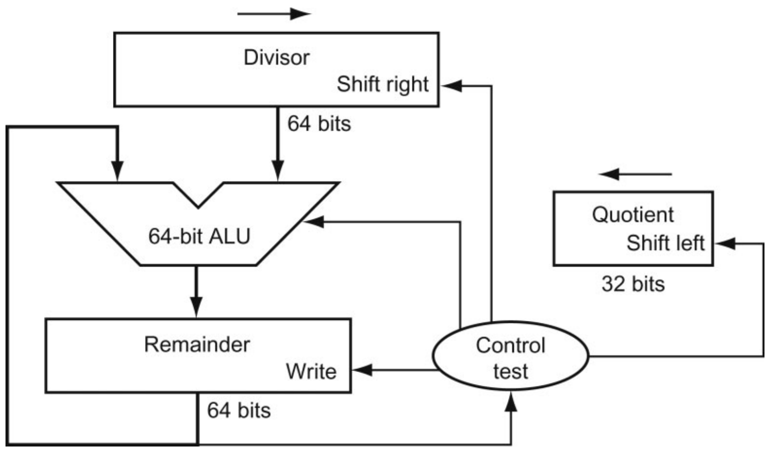
Hardware for Division

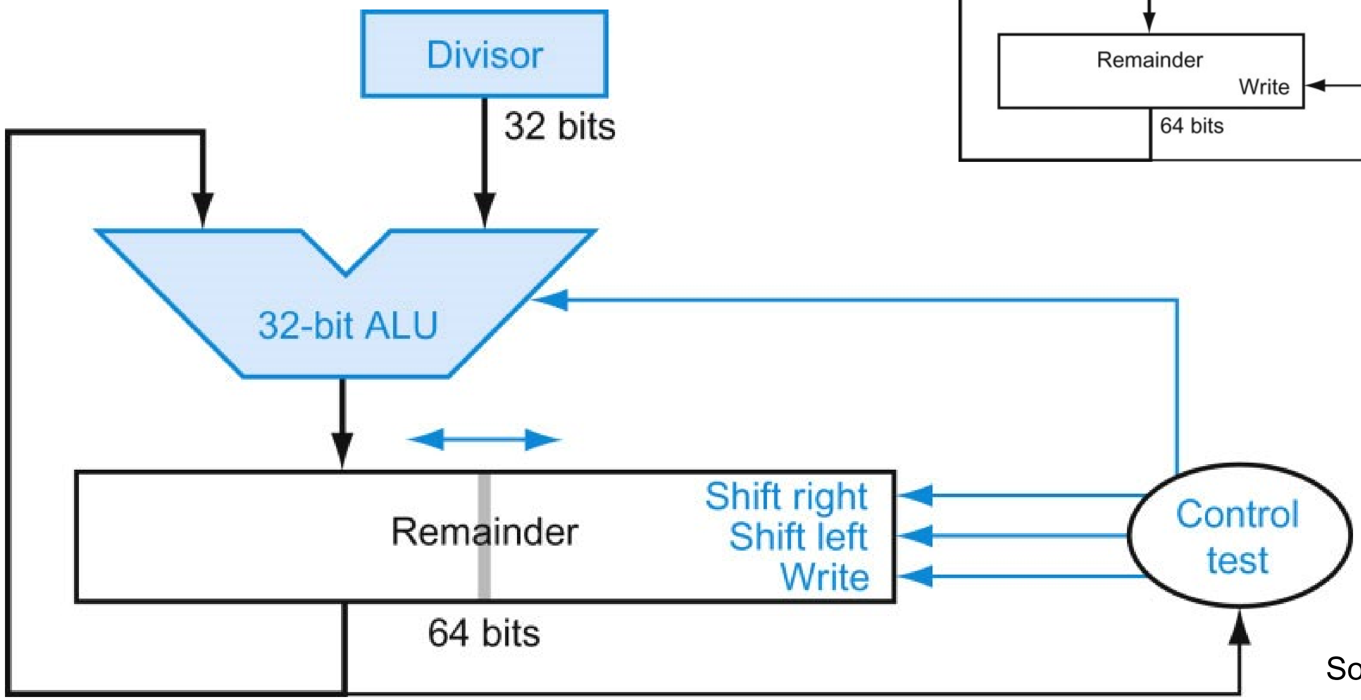


Source: H&P textbook

- การตรวจสอบว่าตัวตั้งมีค่ามากกว่าตัวหารหรือยังจะใช้การลบ ถ้าลบแล้วได้เครื่องหมายเป็นลบ แสดงว่าไม่พอ ก็จะมีการบวกตัวหารกลับเข้าไป

Efficient Division





Source: H&P textbook

Divisions involving negatives

- ในการหารเลขที่เป็นลบนั้น วิธีการที่ง่ายที่สุด คือ เอาเครื่องหมายออกก่อน แล้วค่อยใส่กลับไปทีหลัง

Dividend = Quotient x Divisor + Remainder

+7 div +2	Quo = +3	Rem = +1
-7 div +2	Quo = -3	Rem = -1
+7 div -2	Quo = -3	Rem = +1
-7 div -2	Quo = +3	Rem = -1

- หลักการ : ตัวตั้งและเศษ จะมีเครื่องหมายเหมือนกัน
- ผลลัพธ์จะเป็นลบ ถ้าตัวตั้งและตัวหารมีเครื่องหมายต่างกัน

Exercise

- ให้แสดงการหาร 40_{10} ด้วย 18_{10} โดยแสดงข้อมูลใน register ในแต่ละ step

Iteration	Quotient	Divisor	Reminder
0		0001 0011 0000 0000	0010 1000
1	0	0000 1001 1000 0000	0010 1000
2	00	0000 0100 1100 0000	0010 1000
3	000	0000 0010 0110 0000	0010 1000
4	0000	0000 0001 0011 0000	0010 1000
5	00000	0000 0000 1001 1000	0010 1000
6	000000	0000 0000 0100 1100	0010 1000
7	0000000	0000 0000 0010 0110	0010 1000
8	00000001	0000 0000 0001 0011	10
9	000000010	0000 0000 0000 0100	10

For your attention