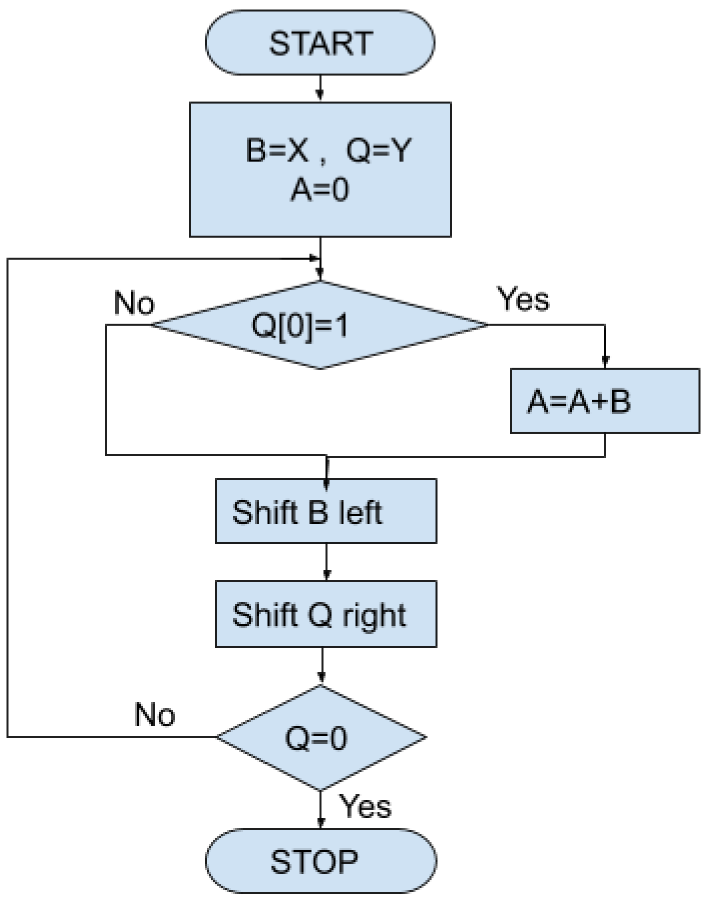
Hardware implementation of various 32-bit and 64-bit multipliers using Bluespec

Done by

M.K.Sanju Vikasini
K.Sreenidhi

Shift and Add Multiplication

- **one_plus_multiplier.bsv**
- **two_plus_multiplier.bsv**
- **three_plus_multiplier.bsv**
- **four_plus_multiplier.bsv**
- **one_adder_multiplier.bsv**
- **two_adder_multiplier.bsv**
- **three_adder_multiplier.bsv**
- **four_adder_multiplier.bsv**



Instructions to use the code:

- Specify the number of bits in the inputs by changing the constant value **'num'**.
- Accordingly change the value of '**num2**' = 2 * **num**.
- Make the above changes in both the **main BSV** file and the "**process_func.bsv**" file.

The Program

- ➔ In **rule r1**, two **num- bit** random numbers are generated and the start method is invoked with one random number (**num-bit**), the other random number (zero extended **(2*num)-bit**) , a 3-bit number and word flag.
  - **Other random number (zero extended (2*num)-bit)-** The number chosen as multiplicand is zero extended to **(2*num)-bit** so that it can initialise the **(2*num)-bit** multiplicand register in which the **'num'** right shift of the multiplicand can be performed.
  - **3-bit number-** If we want to obtain the **lower num-bits** of the (**2*num)-bit** product, the three bit number is **000** and for the **upper num-bits** the number is **011** .
  - **Word flag-** It has to be initialised with **False**.

- ➔ In the **start** method, all the registers are initialised.

- ➔ Then, rule **cycle1** is fired when the multiplier is not equal to zero and rule **cycle2** is fired when multiplier is zero.
  - In rule **cycle2**, the product is assigned 0 .
  - In rule **cycle1,** the complete shift and add operation is performed according to the algorithm.This rule **cycle1** keeps on iterating until the value of the multiplier becomes zero.

- ➔ At this stage, we get the product.

- ➔ The system's answer of the multiplication of the two inputs is calculated in rule **r1** using '*' operator and is stored.

- ➔ In rule **r2**,
  - The **result** method is invoked.
  - In the **result** method, the process function is invoked and this function processes the **(2*num)-bit** product and returns the appropriate num-bits of the product to rule **r2**.
  - The system's answer is compared with that returned by our **result** method.

- ➔ The above steps are executed repeatedly for 'm' times where 'm' is the number of trials specified .

Note:

- ➔ In the **plus variations**, the in-built '+' operator has been used for addition operation in rule cycle1 while in **adder variations**, an explicit full adder has been used.

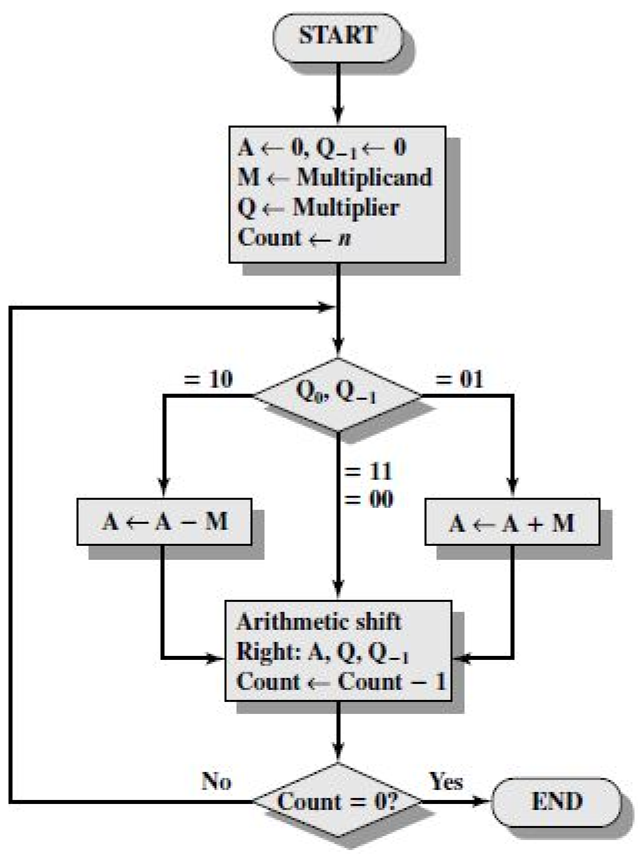
- ➔ The **one,two, three and four** in the different designs, refers to the **number of steps** (iterations in the algorithm) performed in the same cycle.

Booth multiplication

- **booth_iterative.bsv**
- **booth_pipelined_32_2x16.bsv**
- **booth_pipelined_32_4x8.bsv**
- **booth_pipelined_64_4x16.bsv**
- **booth_pipelined_64_8x8.bsv**

★ Iterative

In this algorithm, the inputs are in 2's complement form.That is, this algorithm is for signed multiplication.



Instructions to use the code :

- Specify the number of bits in the inputs by changing the constant value **'num'**.
- Accordingly change the value of '**num2**' = 2 * **num**.
- This code is not integrated with the golden model as the golden model is for processing the outputs of unsigned integer multiplication.

The Program

➔ In rule **r1**, two **num**- bit random numbers are generated and the **inputs** method is invoked with these numbers as arguments.

➔ In the **inputs** method, all the registers are initialised.

➔ Then, rule **rule_mul** is fired when the inputs are ready. In this rule one iteration of the Booth algorithm is performed.

➔ This rule repeatedly fires for **'num'** times(takes **'num'** cycles).

➔ At this stage, we have the final product.

➔ The system's answer of the multiplication of the two inputs is calculated in rule **r1** using '*' operator and is stored.

➔ In rule **r2**,
  - The **outputs** method is invoked, where the (2 * **num**) - bit product is returned .
  - The system's answer is compared with that returned by our **outputs** method.

➔ The above steps are executed repeatedly for 'm' times where 'm' is the number of trials specified .

★ Pipelined

In the pipelined variations, the different iterations in the Booth algorithm are grouped and these groups are instantiated to form the different stages of the pipeline.

For **32-bits**, the 32 iterations are broken down as:

- **2 x 16 -** 2 iterations in a stage and 16 such stages
- **4 x 8 -** 4 iterations in a stage and 8 such stages

For **64-bits**, the 64 iterations are broken down as:

- **4 x 16 -** 4 iterations in a stage and 16 such stages
- **8 x 8 -** 8 iterations in a stage and 8 such stages

Radix-2 multiplication

- **radix_2.bsv**

In this algorithm, the inputs are in 2’s complement form.That is, this algorithm is for signed multiplication.

Algorithm :

➔ Check 2 bits of the multiplier and correspondingly operate on the multiplicand (multiply with 0,1,-1)

➔ The two bits must be grouped in this fashion :
Example : 6 bit number 6 cycles
0 is appended to the right of the LSB of the multiplier.

GrNum
0: 10011**[1(0)]**
1: 1001**[11]**(0)
2: 100**[11]**1(0)
3: 10**[01]**11(0)
4: 1**[00]**111(0)
5: **[10]**0111(0)

Grouped bits

First bit	**Second bit**	**Intermediate term**
0	0	0
0	1	1*multiplicand
1	0	(-1)*multiplicand
1	1	0

➔ Shift this intermediate term (g times - to the right) where, 'g' is the group number(GrNum).
➔ For n-bit number, we will get 'n' such terms.
➔ Add these 'n' terms to get the product (takes n cycles)
➔ In this code we have reduced the number of cycles to n/8 by doing 8 cycle operations in a single cycle.

Instructions to use the code:

- Specify the number of bits in the inputs by changing the constant value **‘num’**.
- Accordingly change the value of ‘**num2**’ = 2 * **num**.
- This code is not integrated with the golden model as the golden model is for processing the outputs of unsigned integer multiplication.

The Program

➔ In rule **r1**, two **num**-bit random numbers are generated and the **inputs** method is invoked with the two random numbers.

➔ In the **inputs** method, all the registers are initialised.

➔ Then, rule **rule_mul** is fired as the inputs are ready.
As per the algorithm, the number of steps required to compute the product is reduced to **num/8** where **num** is the number of bits in the input.
So, in each iteration, the multiplication is done according to the algorithm and rule **rule_mul** iterates **num/8** times.

➔ At this stage, we get the product.

➔ The system’s answer of the multiplication of the two inputs is calculated in rule **r1** using ‘*’ operator and is stored.

➔ In rule **r2**,
  - The **outputs** method is invoked which returns the **(2*num)-bit** product.
  - The system’s answer is compared with that returned by our **outputs** method.

➔ The above steps are executed repeatedly for ‘m’ times where ‘m’ is the number of trials specified .

Radix-4 multiplication

- **radix_4.bsv**

In this algorithm, the inputs are in 2's complement form.That is, this algorithm is for signed multiplication.

Algorithm :

➔ Check 3 bits of the multiplier and correspondingly operate on the multiplicand (multiply with 0,1,2,-1,-2)

➔ The three bits must be grouped in this fashion :
Example : 6 bit number 3 cycles
0 is appended to the right of the LSB of the multiplier.

GrNum
0: 1001**[11(0)]**
1: 10**[011]**1(0)
2: **[100]**111(0)

Grouped bits

First bit	**Second bit**	**Third bit**	**Intermediate term**
0	0	0	0
0	0	1	1*multiplicand
0	1	0	1*multiplicand
0	1	1	2*multiplicand
1	0	0	(-2)*multiplicand
1	0	1	(-1)*multiplicand
1	1	0	(-1)*multiplicand
1	1	1	0

➔ Shift this term (2*g times- to the right) where, g is the group number(GrNum)

➔ For n-bit number, we will get n/2 such terms.

➔ Add these n/2 terms to get the product (takes n/2 cycles)

➔ In this code we have reduced the number of cycles to n/8 by doing 4 cycle operations in a single cycle.

Instructions to use the code:

- Specify the number of bits in the inputs by changing the constant value **‘num’**.
- Accordingly change the value of ‘**num2**’ = 2 * **num**.
- This code is not integrated with the golden model as the golden model is for processing the outputs of unsigned integer multiplication.

The Program

➔ In rule **r1**, two **num**-bit random numbers are generated and the **inputs** method is invoked with the two random numbers.

➔ In the **inputs** method, all the registers are initialised.

➔ Then, rule **rule_mul** is fired as the inputs are ready.
As per the algorithm, the number of steps required to compute the product is reduced to **‘num/8’** where **‘num’** is the number of bits in the input.
So, in each iteration, the multiplication is done according to the algorithm and rule **rule_mul** iterates **‘num/8’** times.

➔ At this stage, we get the product.

➔ The system’s answer of the multiplication of the two inputs is calculated in rule **r1** using ‘*’ operator and is stored.

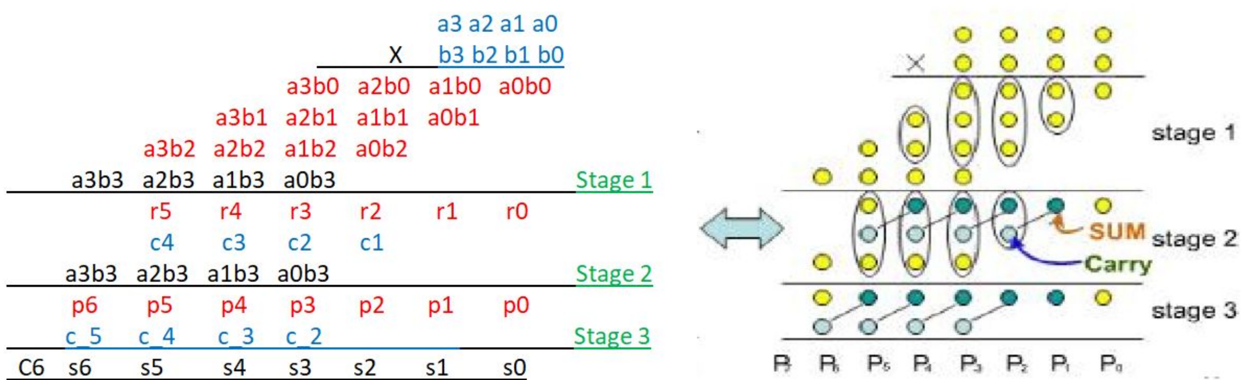
➔ In rule **r2**,
  - The **outputs** method is invoked which returns the **(2*num)-bit** product.
  - The system’s answer is compared with that returned by our **outputs** method.

➔ The above steps are executed repeatedly for ‘m’ times where ‘m’ is the number of trials specified .

Wallace tree multiplication

- **single_cycle_wallace_multiplier.bsv**
- **Multi_cycle_wallace_multiplier.bsv**
- **pipelined_wallace_multiplier.bsv**

★Single-cycle



Algorithm:

- For an n-bit number, **2n partial products** are generated.
- These partial products are **divided into groups** such that each group has **three partial products.**
- As shown in the diagram, the three rows in each group are **added** and each group gives rise to two rows, in the next stage- one is the **sum row** and the other is the **carry row.**
- The rows **left over** in the current stage after the grouping, are **transferred to the next stage.**
- This process of **grouping and adding is repeated** until we reach a stage which has **only 2 rows**.
- In this stage (**last stage**), **ripple addition** of the two rows is carried out, to yield the **2n-bit product.**

Instructions to use the code:

- Specify the number of bits in the inputs by changing the constant value **'num'**.
- Accordingly change the value of '**num2**' = 2 * **num**.
- Make the above changes in both the **main BSV** file and the "**process_func.bsv**" file.

- Change the value of ‘**stages’** according to the value of ‘**num**’.
  Here, stages refers to the number of stages in the computation of product for **num**-bit inputs.

num	**stages**
4	3
8	5
16	7
32	9
64	11

The Program

➔ In rule **r1**, two **num- bit** random numbers are generated and the **start** method is invoked with the two random numbers , a 3-bit number and word flag.
  - **3-bit number-** If we want to obtain the **lower num-bits** of the **(2*num)-bit** product, the three bit number is **000** and for the **upper num-bits** the number is **011** .
  - **Word flag-** It has to be initialised with **False**.

➔ In the **start** method, all the registers are initialised.

➔ Then, the rule **rl1** is fired as the inputs are ready.

➔ In rule **rl1**,
  - The function **fn_mult** is invoked which multiplies two given inputs according to the algorithm and returns the **(2*num)-bit** product.
    - ❖ In **fn_mult**,first,the **‘num’** partial products are generated.
    - ❖ Then, the ‘grouping and adding’ operation is performed **(stages -1)** times so that we are left with just 2 rows.
    - ❖ Finally, the ripple addition on the 2 rows is performed ,producing the result.

➔ At this stage, we get the product.

➔ The system’s answer of the multiplication of the two inputs is calculated in rule **r1** using ‘*’ operator and is stored.

- ➔ In rule **r2**,
  - The **result** method is invoked.
  - In the **result** method, the **process function** is invoked and this function processes the **(2*num)-bit** product and returns the appropriate **num-bits** of the product to rule **r2**.
  - The system's answer is compared with that returned by our **result** method.

- ➔ The above steps are executed repeatedly for 'm' times where 'm' is the number of trials specified .

★Multi-cycle

Instructions to use the code:

- Specify the number of bits in the inputs by changing the constant value **'num'**.
- Make the above changes in both the **main BSV** file and the "**process_func.bsv**" file.

The Program

- ➔ In rule r1, two **num- bit** random numbers are generated and the **start** method is invoked with the two random numbers , a 3-bit number and word flag.
  - **3-bit number-** If we want to obtain the **lower num-bits** of the **(2*num)-bit** product, the three bit number is **000** and for the **upper num-bits** the number is **011** .
  - **Word flag-** It has to be initialised with **False**.

- ➔ In the **start** method, all the registers are initialised.

- ➔ Then, the rule **rl1** is fired as the inputs are ready.

- ➔ In rule **rl1**,the function **fn_partial** is invoked which generates the first three partial products.

- ➔ Then, the rule **rl2** is fired as the partial products are ready.

- ➔ In rule **rl2**,
  - ◆ The function **fn_stage** is invoked.
  - ◆ This function performs addition of the 3 partial product rows in the current stage, hence producing two rows (sum and carry) for the next stage.
  - ◆ Then, the next partial product is generated and inserted as the 3rd row of the next stage.
  - ◆ This rule is fired repeatedly until the last stage (only 2 rows are left) is reached.

- Rule **rl3** is fired when the last stage is reached.
    - In this rule, the function **fn_last_stage** is invoked.
    - This function performs ripple addition on the 2 rows, producing the result.

- Then, rule **rl4** is reached, where the result is extracted and stored.

- The system's answer of the multiplication of the two inputs is calculated in rule **r1** using '*' operator and is stored.

- In rule **r2**,
    - The **result** method is invoked.
    - In the **result** method, the **process function** is invoked and this function processes the **(2*num)-bit** product and returns the appropriate **num**-bits of the product to rule **r2**.
    - The system's answer is compared with that returned by our **result** method.

★Pipelined (32 bit)

Instructions to use the code:

- Specify the number of bits **(32)** in the inputs by changing the constant value **'num'**.
- Make the above changes in both the **main BSV** file and the "**process_func.bsv**" file.

Note:

- The pipelined wallace multiplier, is a variation of the multi-cycle wallace multiplier.

- The different stages of the pipeline -
    - The first stage generates the **first 3 partial products**.
    - The subsequent **7 stages**, each perform **4 steps of the multi-cycle operation**.
    - The **last stage** performs the **last step** of the multi-cycle operation and the **ripple addition**.

- Note : **[3 + (4 * 7) + 1] = 32**