

Module 11 - Structural VHDL

Tutorial and Exercises

For the Mentor Graphics Simulator

Copyright 1995-1999 SCRA

All rights reserved. This information is copyrighted by the SCRA, through its Advanced Technology Institute (ATI), and may only be used for non-commercial educational purposes. Any other use of this information without the express written permission of the ATI is prohibited. Certain parts of this work belong to other copyright holders and are used with their permission. All information contained, may be duplicated for non-commercial educational use only provided this copyright notice and the copyright acknowledgements herein are included. No warranty of any kind is provided or implied, nor is any liability accepted regardless of use.

The United States Government holds “Unlimited Rights” in all data contained herein under Contract F33615-94-C-1457. Such data may be liberally reproduced and disseminated by the Government, in whole or in part, without restriction except as follows: Certain parts of this work to other copyright holders and are used with their permission; This information contained herein may be duplicated only for non-commercial educational use. Any vehicle, in which part or all of this data is incorporated into, shall carry this notice .

See the [RASSP Disclaimer file](#) for additional RASSP Disclaimer, Warranty and Limitation of Liability Information concerning the material, VHDL code and software developed under the RASSP programs or incorporated in RASSP material.

Module 11 Tutorial

1 Getting started

- 1.1 Create a directory for the Module 11 lab material, e.g:

```
mkdir m11_ex
cd m11_ex
```

- 1.2 Copy the source files for the VHDL that you will compile and simulate from the appropriate source directory, e.g:

```
cp $VHDL_SRC/m11_ex/aoi2_str.vhdl
cp $VHDL_SRC/m11_ex/shift_reg8_str.vhdl
```

Note the . at the end of the above commands is important; it tells Unix that you want to copy the file into the same name in the current directory

- 1.3 The Mentor Graphics QuickVHDL simulator needs a work directory for the compiled VHDL files. Create this directory with the appropriate command for the version you are running, e.g:

```
qhlib work
```

- 1.4 The structural modules you will work with in this lab utilize the basic gates you created and compiled for the Module 10 lab. You can access those compiled descriptions by mapping a logical library to their actual location in the file system. The exact command to do this is specific to the VHDL tools being used. For many versions of the Mentor Graphics QuickVHDL simulator, it is done using the following command:

```
qhmap gate_lib ../m10_ex/work
```

This tells the QuickVHDL tools that all modules that are located in the gate_lib logical library can be found in the Unix directory ../m10_ex/work (the “..” is Unix syntax that means “go up one

2 Examine and compile the code for the And-Or-Invert example

- 2.1 Open the file **aoi2_str.vhdl** file using a text editor or a VHDL editing environment. You will see the following VHDL description:

```
-----
--      And/OR Invert Structural Example      --
--      RASSP E&F Module # 11 Structural VHDL --
--      Robert Klenke UVa 19 April 1996      --
-----

LIBRARY gate_lib;
```

```

USE gate_lib.resources.all;

ENTITY aoi2_str is
  GENERIC(trise : delay := 12 ns;
          tfall : delay := 9 ns);
  PORT(a : IN level;
        b : IN level;
        c : IN level;
        d : OUT level);
END aoi2_str;

ARCHITECTURE structural OF aoi2_str IS

  COMPONENT and2
    GENERIC(trise : delay;
            tfall : delay);
    PORT(a : IN level;
          b : IN level;
          c : OUT level);
  END COMPONENT;

  COMPONENT or2
    GENERIC(trise : delay;
            tfall : delay);
    PORT(a : IN level;
          b : IN level;
          c : OUT level);
  END COMPONENT;

  COMPONENT inv
    GENERIC(trise : delay;
            tfall : delay);
    PORT(a : IN level;
          b : OUT level);
  END COMPONENT;

  FOR ALL : and2 USE ENTITY gate_lib.and2(behav);
  FOR ALL : or2  USE ENTITY gate_lib.or2(behav);

  SIGNAL and_out : level;    -- signal for output of and gate
  SIGNAL or_out  : level;    -- signal for output of or gate

  BEGIN

    AND_1 : and2 GENERIC MAP(trise => trise, tfall => tfall)
      PORT MAP(a => a, b => b, c => and_out);

    OR_1  : or2  GENERIC MAP(trise => trise, tfall => tfall)
      PORT MAP(a => and_out, b => c, c => or_out);

    INV_1 : inv  GENERIC MAP(trise => trise, tfall => tfall)
      PORT MAP(a => or_out, b => d);

  END structural;

```

Notice that the **and2**, **or2**, and **inv** gates are bound to the components in the library **gate_lib** that you mapped to the Module 10 examples earlier.

2.2 Compile the VHDL code, e.g:

```
qvhcom aoi2_str.vhdl
```

aoi2_str.vhdl should compile without any errors. The compiler should display a message similar to the following with no errors:

```
// Compiling for QuickHDL
// QuickHDL qvhcom v8.5_4.5a Mar 28 1996 SunOS 4.1.3
//
// Copyright (c) Mentor Graphics Corporation, 1982-1995, All Rights Reserved.
//      UNPUBLISHED, LICENSED SOFTWARE.
//  CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//  PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
//
// Copyright (c) Model Technology Incorporated 1990-1995, All Rights Reserved.
//
-- Loading package standard
-- Loading package resources
-- Compiling entity aoi2_str
-- Compiling architecture structural of aoi2_str
-- Loading entity and2
-- Loading entity or2
-- Loading entity inv
```

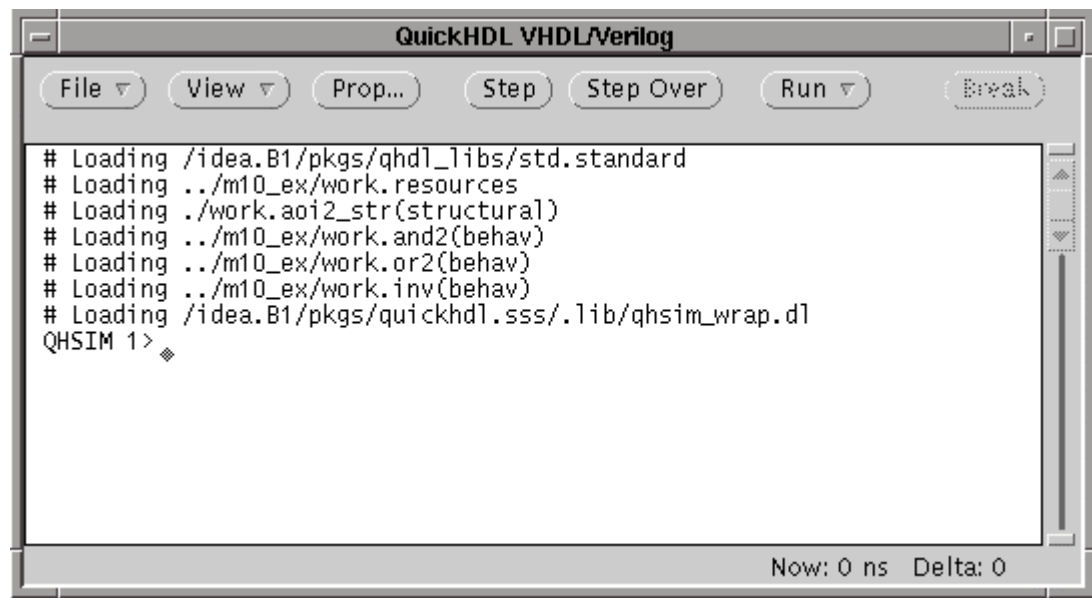
Notice that during compilation that the entities for the **and2**, **or2**, and **inv** components were loaded by the compiler to check that they properly matched the generic and port maps used in the component declarations and instantiations. Only the entities were checked by the compiler, the architectures are not needed until the design is actually simulated and are not checked during compilation.

3 Simulate the compiled code

3.1 Start up the Mentor Graphics VHDL simulator. The specific command may vary depending on the version you are using, e.g.:

```
qhsim aoi2_str
```

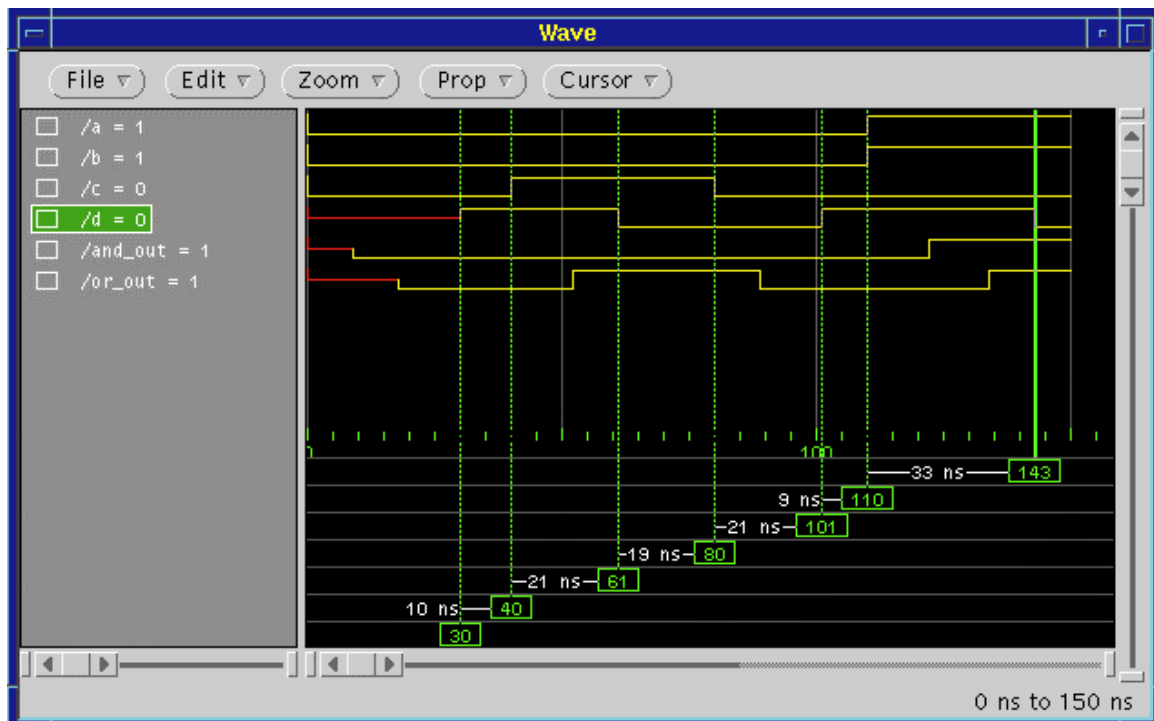
This will bring up a window similar to the one shown below. Notice that the architectures for the **and2**, **or2**, and **inv** gates are loaded at this time. If there were any problems with the architectures it will show up here:



- 3.2 Next, select signals ***a***, ***b***, and ***c*** for viewing. Consult the documentation for your specific simulator version for instructions on selecting signals for viewing. Use the *force* mechanism to set values for the input signals and run the simulation for 150 ns, e.g:

```
QHSIM 1> force -freeze /a 0, 1 110
QHSIM 2> force -freeze /b 0, 1 110
QHSIM 3> force -freeze /c 0, 1 40, 0 80
QHSIM 4> run 150
```

After adding cursors, the resulting window should look similar to this:



4 Examine and compile the code for the 8 bit shift register example

- 4.1 Open the file **shift_reg8_str.vhdl** file using a text editor or a VHDL editing environment.
- 4.2 Compile the VHDL code. **shift_reg8_str.vhdl** should compile without any errors, e.g:

```
qvhcom shift_reg8_str.vhdl
```

5 Simulate the compiled code

- 5.1 Start up the Mentor Graphics VHDL simulator. The specific command may vary depending on the version you are using, e.g.:

```
qhsim shift_reg8_str
```

- 5.2 Next, select signals all the signals in the component for viewing. Consult the documentation for your specific simulator version for instructions on selecting signals for viewing. Use the *force* mechanism to set values for the input signals and run the simulation for 300 ns, e.g:

```
QHSIM 5> force -freeze /clk 0 -repeat 40
```

```
QHSIM 6> force -freeze /clk 1 20 -repeat 40
```

```
QHSIM 7> force -freeze /enable 1
```

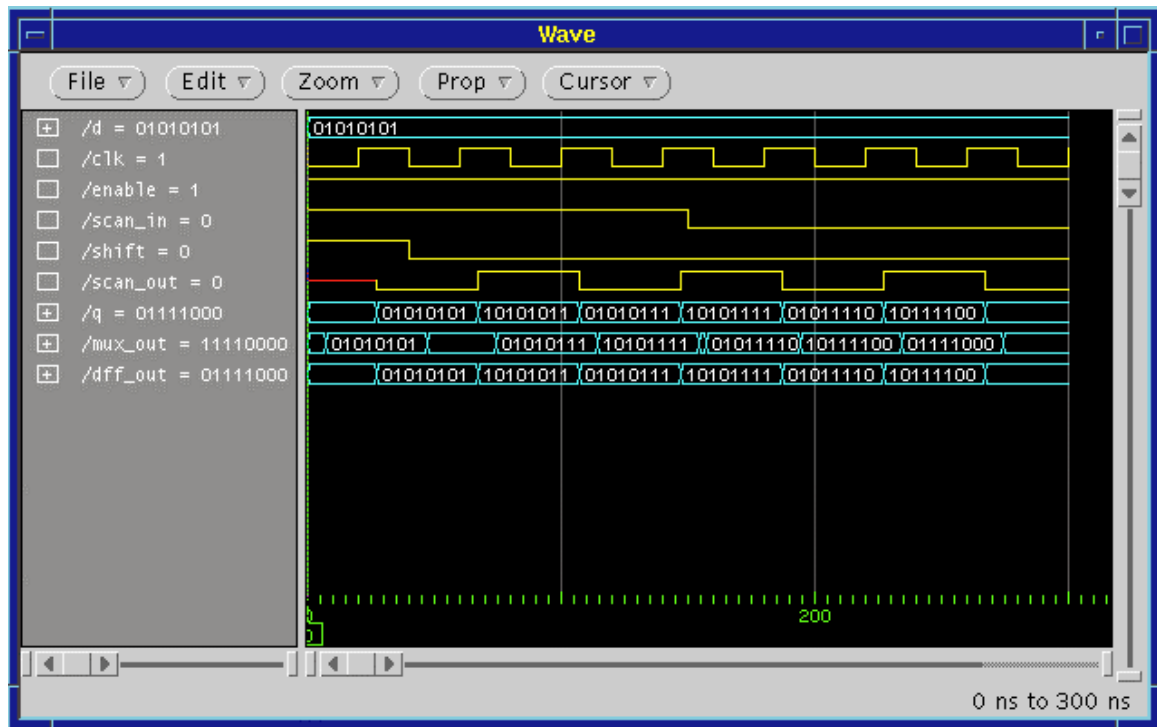
```
QHSIM 8> force -freeze /d 01010101
```

```
QHSIM 9> force -freeze /shift 1, 0 40
```

```
QHSIM 10> force -freeze /scan_in 1, 0 150
```

```
QHSIM 11> run 300
```

The resulting window should be similar to this:



6 Copy and compile the code needed for the unsigned 8 bit multiplier example

6.1 Copy the VHDL files from the source directory, e.g:

```
cp $VHDL_SRC/m11_ex/reg8_str.vhdl .
cp $VHDL_SRC/m11_ex/ha_str.vhdl .
cp $VHDL_SRC/m11_ex/fa_str.vhdl .
cp $VHDL_SRC/m11_ex/alu_str.vhdl .
cp $VHDL_SRC/m11_ex/mult_datapath_str.vhdl .
```

6.2 Compile the VHDL files, e.g:

```
qvhcom reg8_str.vhdl
qvhcom ha_str.vhdl
```

```
qvhcom fa_str.vhdl
```

```
qvhcom alu_str.vhdl
```

```
qvhcom mult_datapath_str.vhdl
```

- 6.3 There is also a command file (called a *do* file) that has been created that will generate the forces necessary to drive the datapath through an example multiplication sequence. Copy it over now from the source directory, e.g:

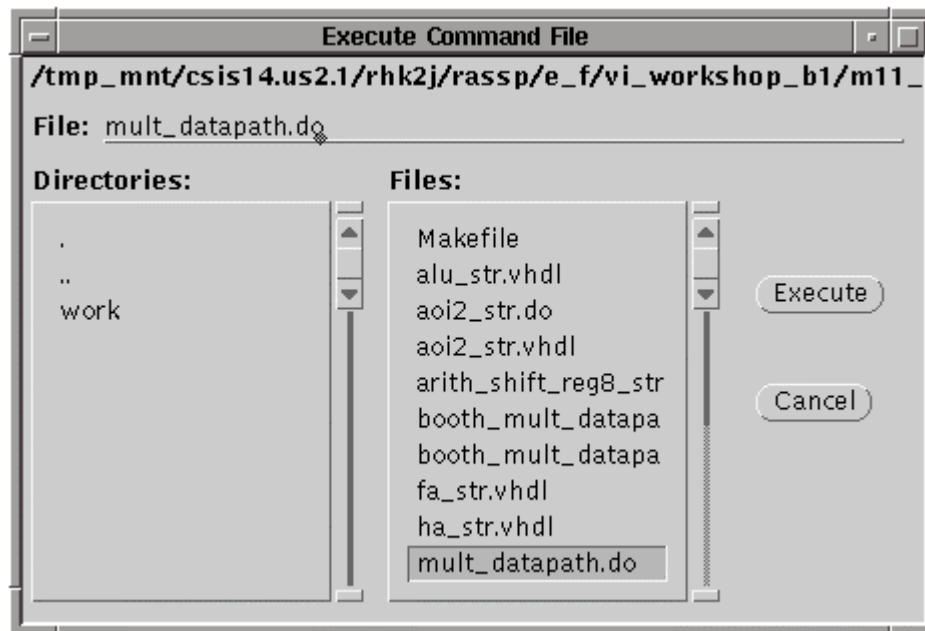
```
cp $VHDL_SRC/m11_ex/mult_datapath.do .
```

7 Simulate the unsigned 8 bit multiplier datapath example

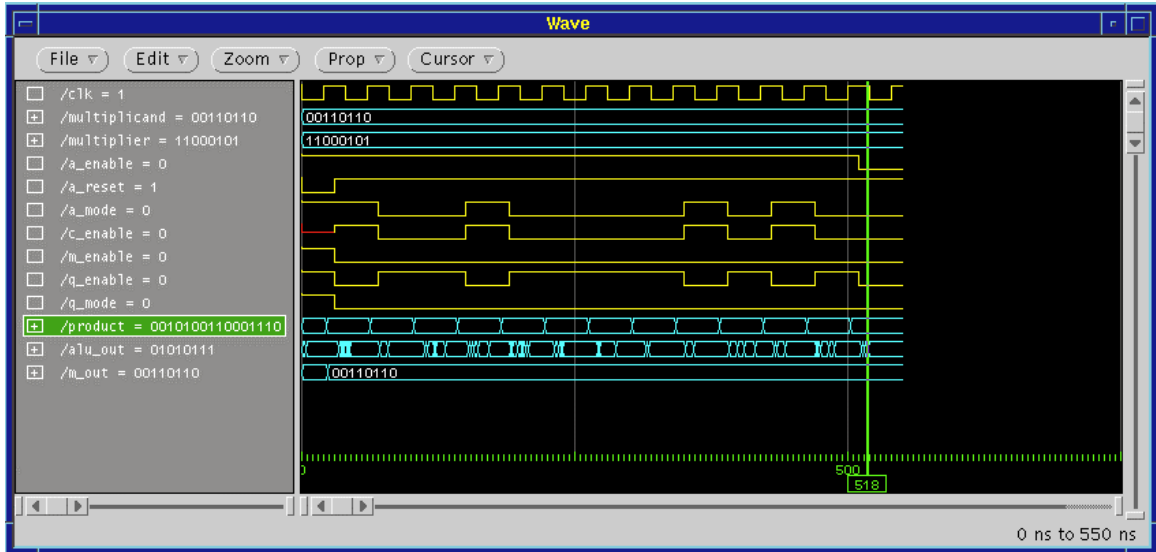
- 7.1 Start up the Mentor Graphics VHDL simulator. The specific command may vary depending on the version you are using, e.g.:

```
qhsim mult_datapath
```

- 7.2 You can use the *dofile* mechanism in the Mentor Graphics VHDL simulator to facilitate the setup of a simulation, e.g:



Executing the `mult_datapath.do` file should result in a window similar to this:



Notice that during initialization, the ALU outputs all “0”s which is loaded into the A register to clear it, The C register is cleared, the multiplier is loaded into the Q register and the multiplicand is loaded into the M register. The simulation then proceeds by performing ADD and SHIFT operations based on the status of the Q0 bit.

Module 11 Exercise

Assignment:

Using a simple GENERATE statement, create an 8 bit 2 to 1 multiplexor from a structure of **mux2** components. Compile and simulate the design to ensure correct operation. Do the same for an 8 bit 4 to 1 multiplexor using **mux4** components.

Using a GENERATE statement with an IF clause, create an 8 bit arithmetic shift register from **dff** and **mux2** components. Recall that in an arithmetic shift:

$Q(7) \leq D(7),$

$Q(6 \text{ downto } 0) \leq D(7 \text{ downto } 1),$ and

$\text{Scan_out} \leq D(0).$

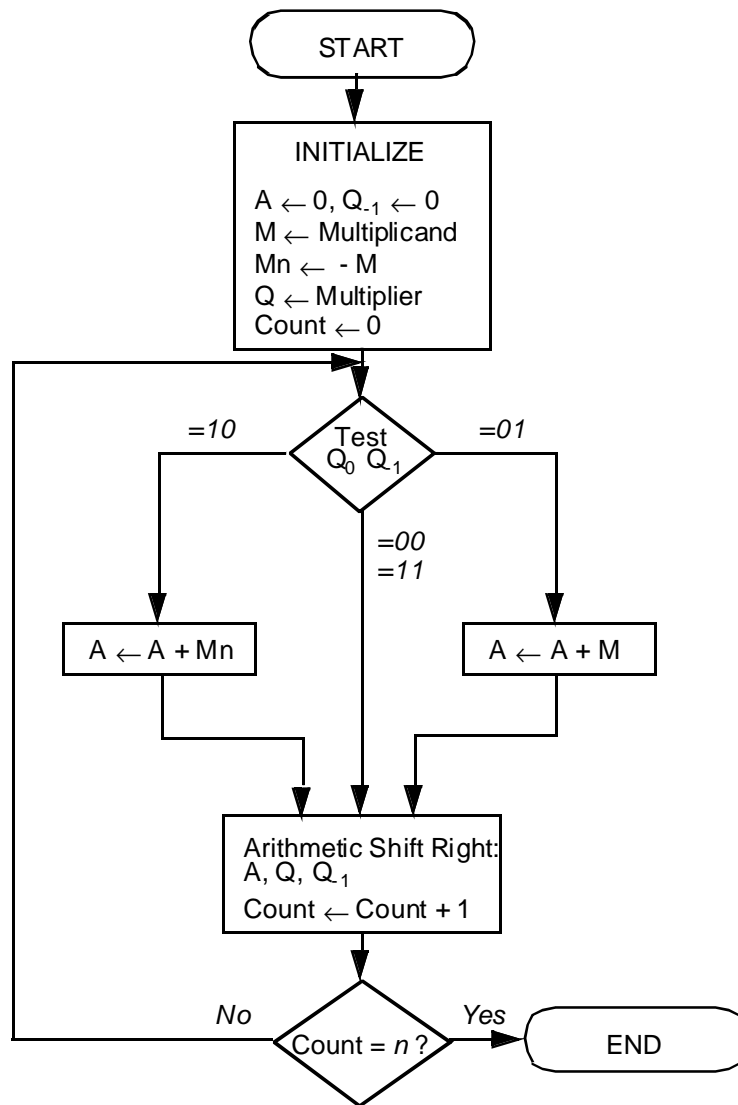
Compile and simulate the design to ensure correct operation.

Using the components constructed above as well as the **reg8_str**, **shift_reg8_str**, **alu_str**, **mux2**, and **dff**, create an RTL datapath for a Booth's algorithm multiplier.

A Booth's multiplier functions very similarly to the unsigned multiplier with the exceptions that it uses an arithmetic shift of the A and Q registers, the Q(0) bit is shifted out into a Q(-1) register, and the control of the shift/add operation is based on the Q(0) and Q(-1) bits. Finally instead of just adding M to A and shifting, The Booth's multiplier either adds or subtracts (adds the two's complement of M) with A. The flow chart on the next page fully outlines Booth's algorithm for two's complement multiplication.

An example of 2-complement multiplication using Booth's Algorithm is shown below:

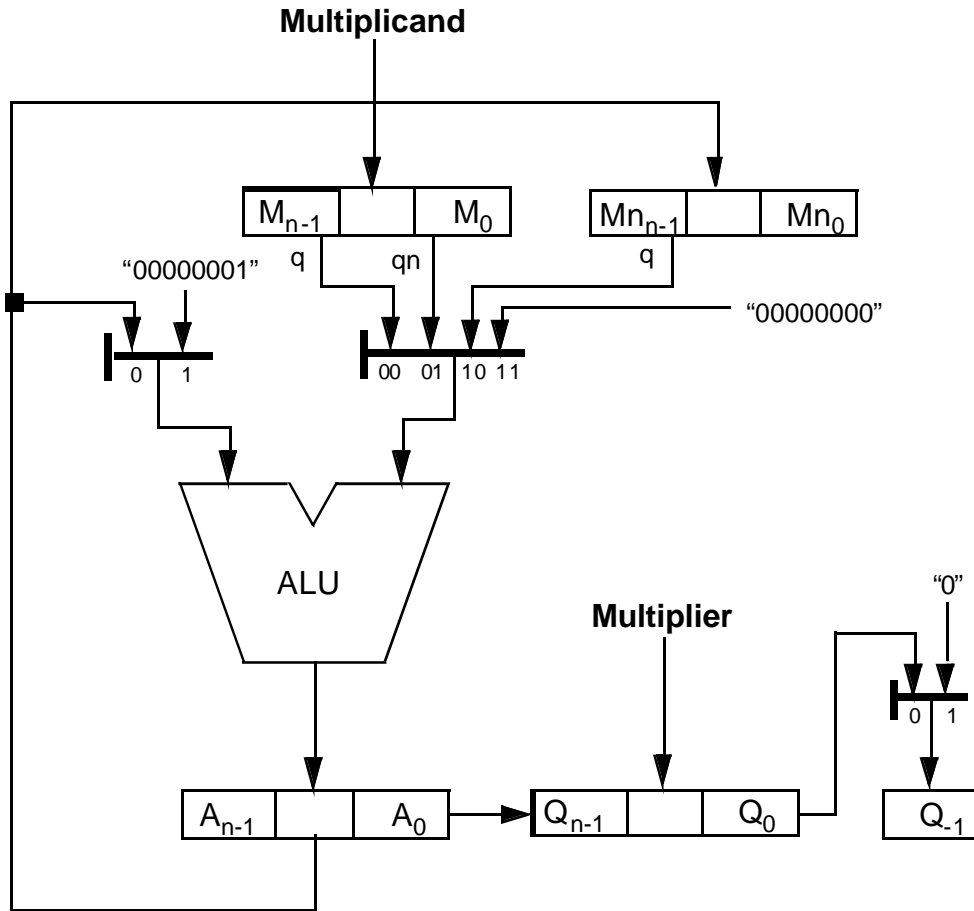
M = 11111011		Mn = 00000101			
A	Q	Q ₋₁			
00000000	00000011	0	Initialization		
00000101	00000011	0	$A \leftarrow A + Mn$	}	First Cycle
00000010	10000001	1	Arithmetic Shift		
00000001	01000000	1	Arithmetic Shift	}	Second Cycle
11111100	01000000	1	$A \leftarrow A + M$		
11111110	00100000	0	Arithmetic Shift	}	Third Cycle
11111111	00010000	0	Arithmetic Shift		
11111111	10001000	0	Arithmetic Shift	}	Fourth Cycle
11111111	10001000	0	Arithmetic Shift		
11111111	11000100	0	Arithmetic Shift	}	Fifth Cycle
11111111	11000100	0	Arithmetic Shift		
11111111	11100010	0	Arithmetic Shift	}	Sixth Cycle
11111111	11100010	0	Arithmetic Shift		
11111111	11100010	0	Arithmetic Shift	}	Seventh Cycle
11111111	11110001	0	Arithmetic Shift		
11111111	11110001	0	Arithmetic Shift	}	Eighth Cycle
11111111	11110001	0	Arithmetic Shift		



Flowchart for Booth's Algorithm 2's-Complement Multiplication

Hint - A datapath very similar to the one used for the unsigned multiplier can be used. Instead of a single register entering the ALU for the Multiplicand, two registers can be used, one for the multiplicand (M) and another for the two's complement of M (Mn) entering the ALU through a 4 to 1 multiplexor. The two's complement of M can be generated by adding the inverse of M (available as the Qn outputs of the M register) to "00000001" using the ALU.

If you need help, an example datapath that can perform Booth's Algorithm multiplication is shown on the following page.



Example Datapath for Booth's Algorithm 2's-Complement Multiplication