DESIGN PROJECT 2 – BCD CALCULATOR – Part 2: Control Design

ECE/Comp Sci 352 – Digital System Fundamentals – Semester II 1998-99

This part to be submitted at the same time as Part 1. Overall Project: Due Class Time, Friday, April 30, 1999; 15% of course grade.

Please see the introductory section of Part 1 for the general project guidelines. Note that there is an additional Team Effort Report included with this part of the project to be submitted with your project report.

In this part, we will guide you through the design of the control of the calculator and the integration and testing of the datapath and the control. **You are strongly encourage to study this entire write-up before beginning. Also, it is very important that you read your e-mail to keep up to date on specification changes, etc.**

EXTERNAL SPECIFICATIONS

The external specifications are repeated here for your reference. The BCD Calculator user interface is shown in Figure 1; it is modeled after a Texas Instruments™ TI™-1768II calculator. Differences are that it has only four digits, handles only integers, and lacks multiplication, division, memory, and an error indicator. In addition to the **Power/Reset** button, the calculator has 10 digit buttons, **0** through **9**, for entering decimal digits, and four operation buttons, **Clear**, **+**, **–,** and **=.** There is a 4 and 1/2 digit LCD display for displaying operands, intermediate values and signed results. The 1/2 digit is a minus sign. Entered operands are non-negative.

The calculator has the following properties:

1. It uses “infix” notation, e. g., entry sequence 5, + , 6, –, 7 gives (5 + 6) – 7 = 4.
2. When the first digit of an operand is entered, the most significant three digits go to 0.
3. If more than four digits are entered, the last four digits are used as the operand.

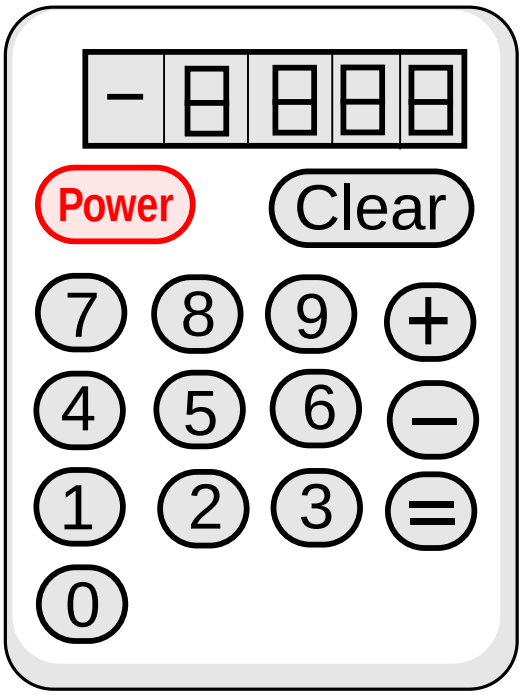


Figure 1: BCD Calculator Appearance

WARNING! SPECIFICATION CHANGE Explicit specification of the immediate execution of Clear! Comment on =.

4. If a succession of operators is entered without entering any digits, the last of the operator entered is the one executed with the exception of Clear which is executed immediately upon entry. Impicitly, the = also executes immediately since it causes the prior operation to be executed.
5. The CLEAR operator clears the result register and the entry register and attaches the entry register to the display.
6. The +, – operators perform their operations just once. If they are repeated before entering new digits, no operation is executed.

WARNING! SPECIFICATION CHANGE To simplify your design, specification 7 has been deleted!

7. ~~In contrast, when the = is pressed multiple times, the last computation is repeated, e.g., 5, +, 6, =, =, = gives ((5+6) + 6) +6 = 23.~~
8. There is no error or overflow indicator.

WARNING! SPECIFICATION CHANGE An additional feature of the execution of = was discovered.

9. If the = operator is followed by a digit entry, then a new calcuation is started. This is accomplished by clearing the ACC_R.
10. If the = operator is followed by an operator entry, then the computation continues without clearing the ACC_R.

INTERNAL SPECIFICATIONS

The top level of the design of the BCD Calculator appears in Figure 2.

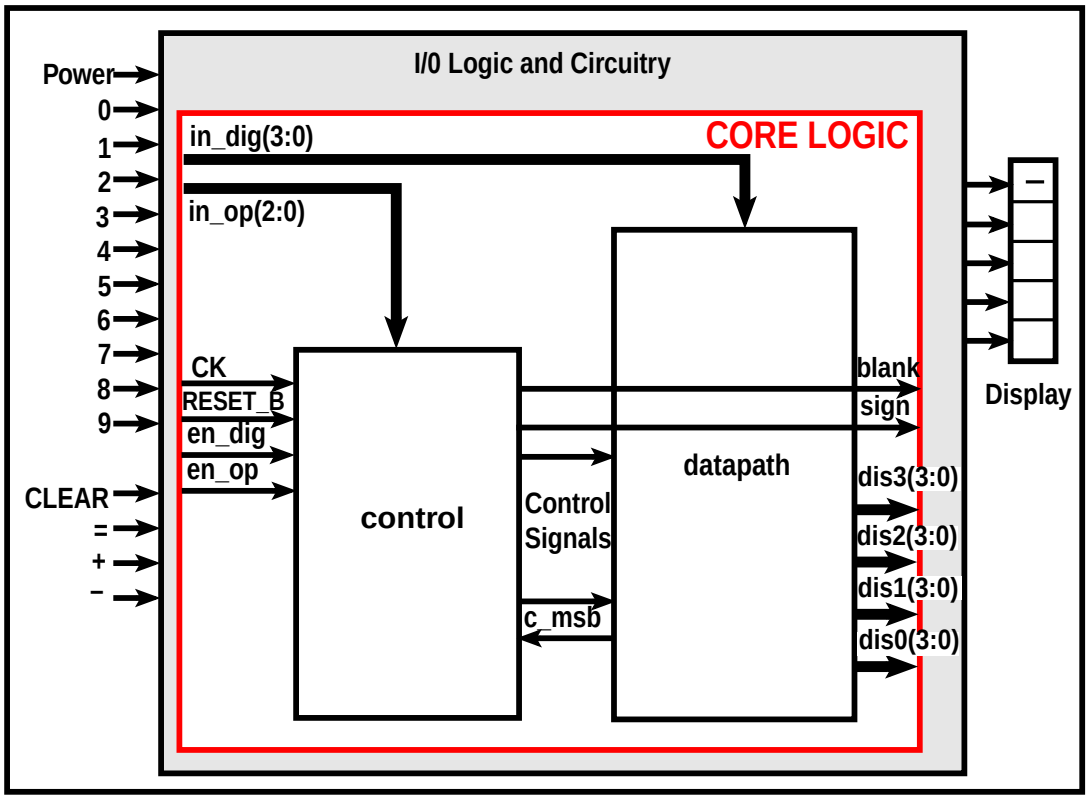


Figure 2: Top Level Design

You are to design only the **CORE LOGIC**! The **I/O Logic and Circuitry** given in Part 1 is specified functionally as follows and is **not** to be designed! This description of the interface to the outside world will permit you to understand the environment for the **CORE LOGIC**.

The external pushbuttons and the display connect to the I/O Logic and Circuitry which does the following:

1. provides a clock signal, **CK**.
2. provides a master reset signal, **Reset**, that is activated when the power is turned on.
3. debounces and otherwise conditions signals from the pushbuttons.
4. for each decimal digit pushbutton provides, when the button is pushed:

   a. **en_dig** = 1; otherwise, **en_dig** = 0.

   b. **in_dig(3:0)**, the 4-bit BCD representation for the decimal digit, both for one clock cycle,

5. for each operation pushbutton provides, when the button is pushed:

   a. **en_op** = 1, otherwise, **en_op** = 0.

   b.**in_op(2:0)**, both for one clock cycle (The 3-bit representation for the operation defined as:

Clear: 100, =: 010, –: 001, and +: 000.), and

WARNING! New output signal: blank.

6. takes the 4-digit output **dis3(3:0)**, **dis2(3:0), dis1(3:0),** and **dis0(3:0)** and **sign** from the **CORE LOGIC** and provides the display drivers for driving the 4-1/2 digit display. **blank** shuts off the display drivers when operations are being performed and the intermediate output values are of no interest to the user.

CALCULATOR CORE

The schematic for the calculator core appears in Figure 3. This shows the inputs and

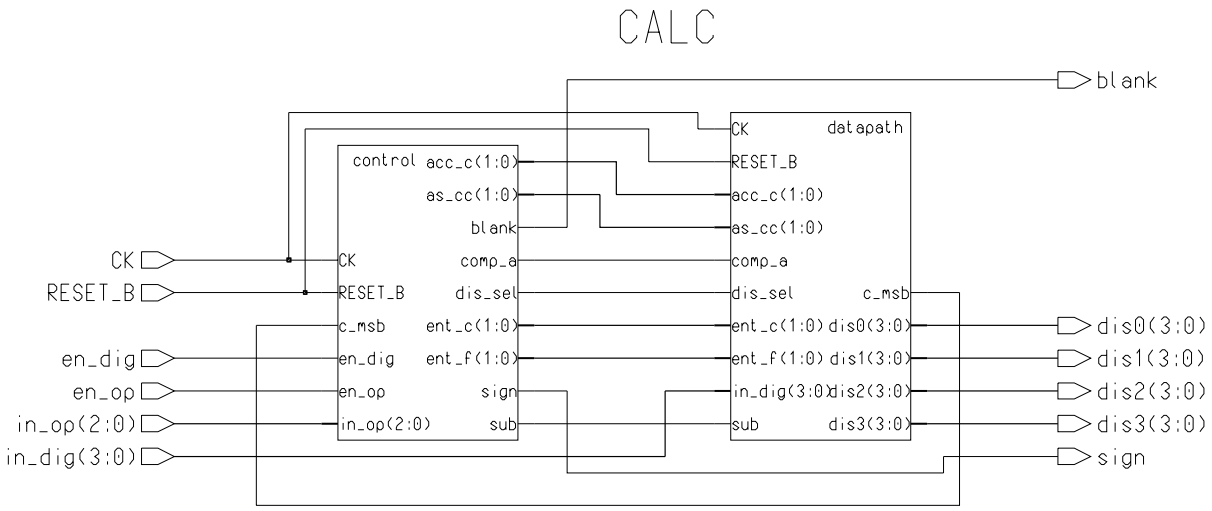


Figure 3: Core Schematic

outputs for the core and shows the control and status signals between the control and the datapath.

DATAPATH CONTROL SPECIFICATIONS

A revised schematic for the **datapath** appears in Figure 4.

WARNING! An engineering change has been made in the datapath. Incorporate this change in your datapath.

Revision: The value change on **c_msb** was one clock cycle too late to work with the control design. So **c_msb** has been moved from the output **CQ** of **carry** to the output **COUT** of **bcd_digit**.

The tables for the control inputs to the datapath are given here to assist you in understanding and defining the output values of the control that determine the microoperations performed.

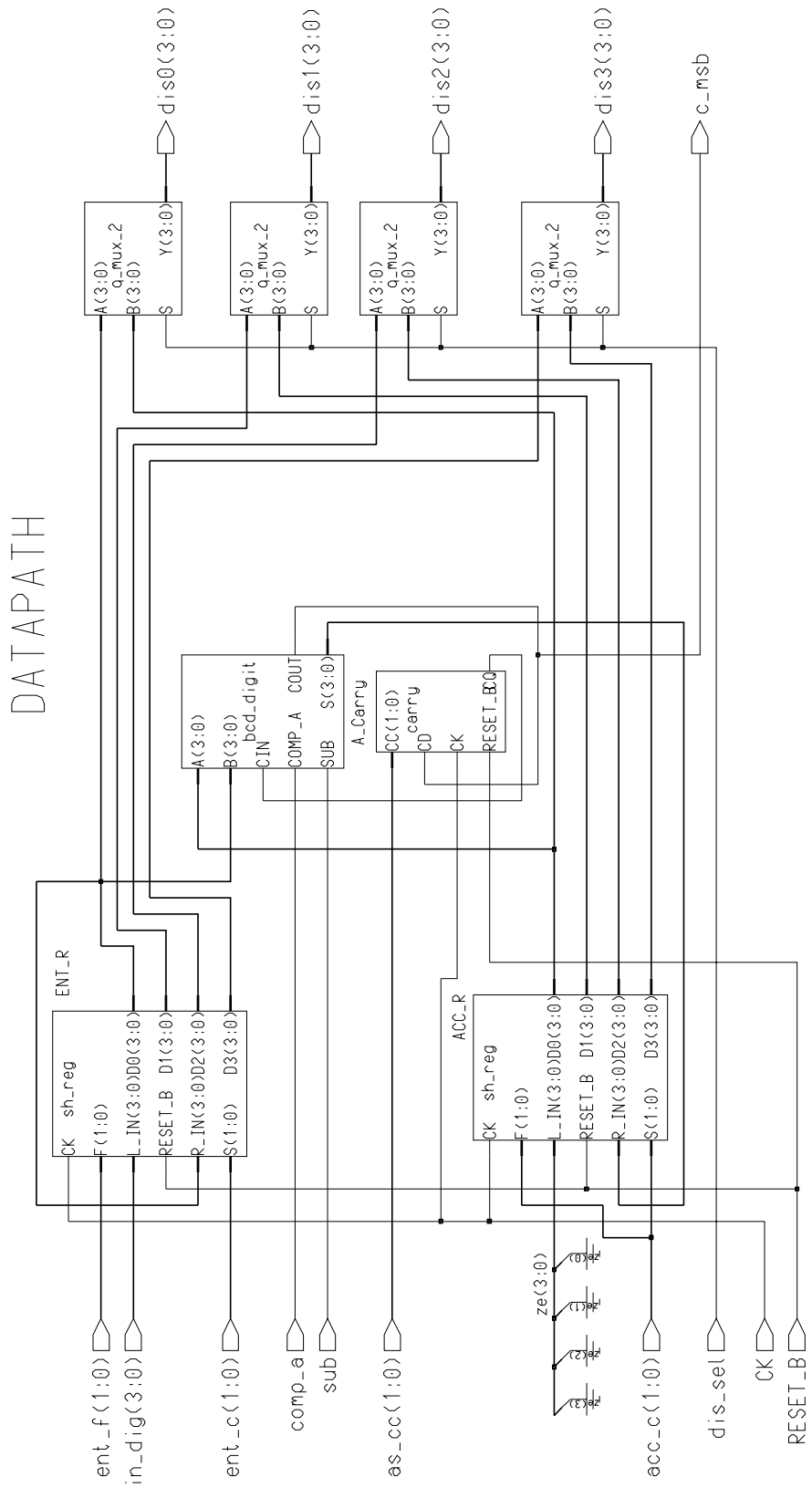


Figure 4: datapath

bcd_digit. Component **bcd_digit** performs BCD digit addition, subtraction, and complementation. Its operation is defined in the following table:

bcd_digit			
OP NAME	**COMP_A**	**SUB**	**OPERATION**
add	0	0	BCD Addition: (COUT, S) = A + B + CIN [in BCD]
sub	0	1	BCD Subtraction: (COUT, S) = A + 9's comp (B) + CIN [in BCD]
comp10s	1	0	BCD 10"s Complementation (COUT, S) = 9's comp(A) + CIN [in BCD}

sh_reg. The operation of **sh_reg** is described by the following table:

sh_reg		
OP NAME	**F(1:0) OR S(1:0)**	**OPERATION**
hold	0 0	Hold the contents unchanged
sr	0 1	Shift all digits one position to the right
sl	1 0	Shift all digits one position to the left
clr	1 1	Clear the contents to all 0 digits

Note that there are two control fields **F(1:0)** and **S(1:0)**. **F(1:0)** controls the least significant digit position **D0(3:0)** and **S(1:0)** controls the other three digit positions. This is because of one combined operation used when the first digit is entered in **ENT_R**. For this operation, **F(1:0)** = 10 and **S(1:0)** = 11, loading the value on **L_IN** into **D0(3:0)** and clearing to 0 the rest of the digit positions. Otherwise, **F(1:0)** and **S(1:0)** are set to the same values.

carry. The operation of carry is described by the following table:

carry		
OP NAME	**CC(1:0)**	**OPERATION**
hold	0 0	Hold the contents of the flip-flop with output COUT unchanged
reset	0 1	Reset the flip-flop with output COUT to 0
set	1 0	Set the flip-flop with output COUT to 1
load	1 1	Load value on CIN into the flip-flop with output COUT

q_mux_2. This is a straightforward quad 2-to-1 multiplexer. Its operation is described by the following table:

q_mux_2		
OP NAME	**S**	**Operation**
dis_ent	0	Select input A(3:0)
dis_acc	1	Select input B(3:0)

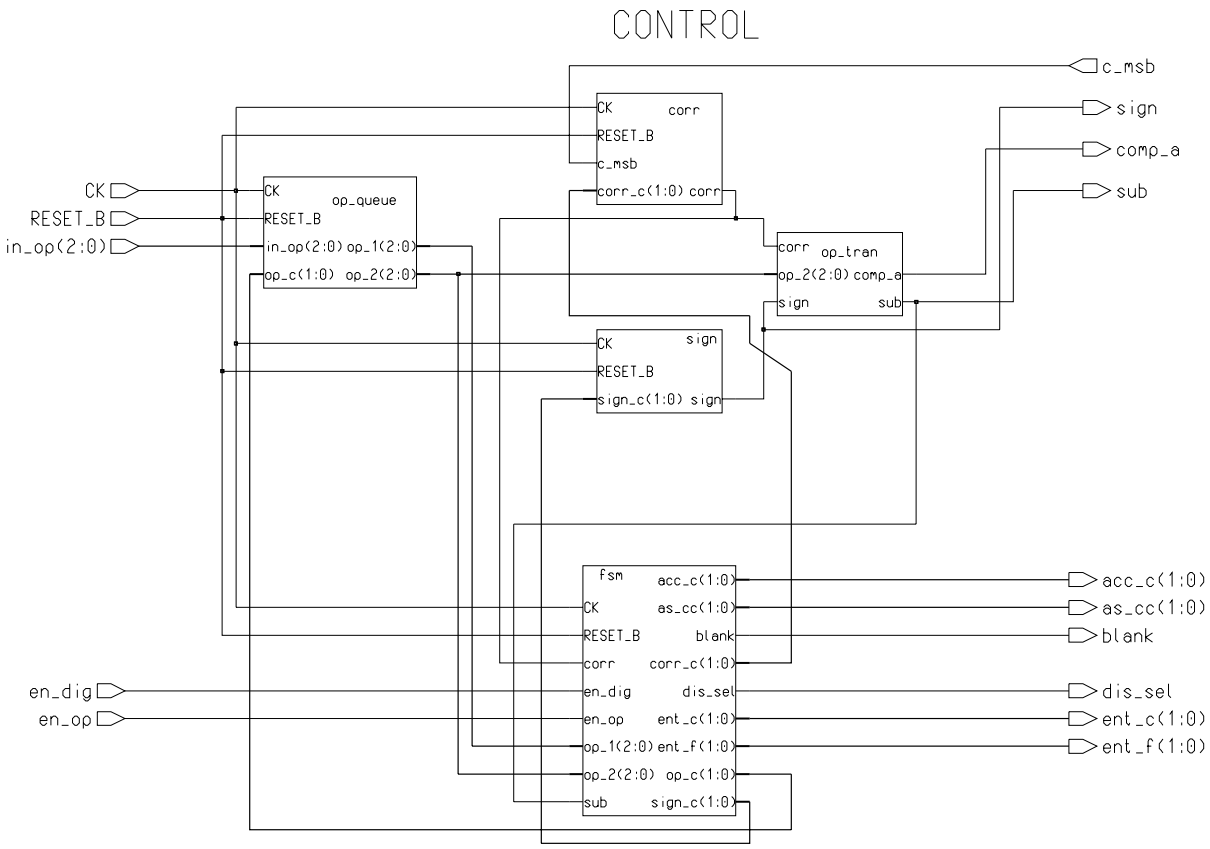
DATAPATH OPERATION

To aid in understanding the control, we will briefly go over the **datapath** operation with reference to Figure 4. The digits of an operand are entered most significant bit first. Thus, it is appropriate to use left shift (sl) of the entry register **ENT_R** for digit entry. For example for entry of operand 123, 1 is shifted through **L_IN** into the rightmost digit position of **ENT_R**, **D0(3:0)**. Next, 2 is shifted into **D0(3:0)** and the 1 is moved to the left from **D0(3:0)** to **D1(3:0)**. Then 3 is shifted through **L_IN** into **D0(3:0)**, the 2 is shifted from **D0(3:0)** to **D1(3:0)** and the 1 is shifted from **D1(3:0)** to **D2(3:0)**. At the end of the operand entry, **ENT_R** contains 0123. Next, an operation, say + is entered. This causes the operand in **ENT_R** to be moved to **ACC_R**. This movement is accomplished by serial addition of the contents to of **ENT_R** to the contents of **ACC_R** which has been initialized to zero. The addition is done serially, least significant digits first. This requires right shift (sr) of **ENT_R** and **ACC_R**. Initially, **D0(3:0)** of **ENT_R** and **D0(3:0)** are presented on inputs **A(3:0)** and **B(3:0)**, respectively of **bcd_digit** which produces the digit sum and digit carry on **S(3:0)** and **COUT**, respectively. When the **CLK** occurs, **ENT_R** and **ACC_R** are shifted right, moving the contents of **D1(3:0)** in each register to the **D0(3:0)** position in preparation for the next digit addition. The result of the first digit addition from **S(3:0)** is shifted into the most significant end of **ACC_R**. Finally, the digit carry is stored in **A_Carry** for use in the next digit addition. After repeating this same action three more times, the sum of the contents of **ENT_R** and 0 appears in **ACC_R**.

This same sequence of four right shift actions is used to perform addition and subtraction. However, it is necessary to initialize the carry appropriately before the operation. For an addition, **A_Carry** is initialized to 0 and for a subtraction, **A_Carry** is initialized to 1. BCD addition (op_2 = 000) and BCD subtraction (op_2 = 001) are sign-magnitude operations. One operand has its sign in flip-flop **sign** in the control and its magnitude in **ACC_R**. The second, implicitly positive operand is held in **ENT_R**. Subtraction is **ACC_R** – **ENT_R** = **ACC_R** + 9's comp (**ENT_R**) + 1 and uses the required 10's complement correction if the carry from the msb (**c_msb**) of the subtraction is 0. This requires storage of an indicator that **c_msb** was 0 and a 10's complement operation be performed on **ACC_R** which will contain the initial uncorrected result. The flip-flop storing $\overline{\textbf{c\_msb}}$ is **corr**. In sign magnitude, it is necessary to determine the operation **add** or **sub** to actually be performed in the datapath based on the operation specified in **op_queue** (actually bit 0 in **op_2**). This function is performed by a block called **op_tran** for operation translation. This block generates the control signal for the 10's complement and the sign complement and gives the complement operation priority over subtraction.

CONTROL SPECIFICATIONS

A schematic diagram of the overall **control** is given in Figure 5. **control** contains operation queue **op_queue**, correction flip-flop **corr**, operation translation logic **op_tran**, sign flip-flop **sign**, and state machine **fsm** with state sequencing (next state) and output



logic. The function of each of the components of the control will be described next in detail.

op_queue. Because of the use of infix notation in the calculator, an operation may be executed immediately or may be executed only after a second operand is entered. For example, a **Clear** is executed before any additional digits are entered into the calculator. On the other hand, a + is executed only after the second operand following the + has been entered and an additional operation has also been entered. Thus, in the case of the +, it must be saved (queued) for execution until after the next operation, e. g., –, has been entered. Thus, there must be a capability for storing two operations if either + or – is the first operation.

op_queue consists of two 3-bit registers wired so that the top register **op_1** serves as the input to the bottom register **op_2**. The input to **op_1** is **in_op(2:0),** the operation code from the I/O logic, **in_op(2:0)**. The outputs of both registers are available for use by the rest of the **control**.

The functions performed by **op_queue** are given in the following table:

op_queue		
OP NAME	**op_c(1:0)**	**OPERATION**
hold	0 0	Hold the contents of registers op_1 and op_2 unchanged
reset	01	Reset the contents of op_1 and op_2 to all 0's
advance	10	Simultaneously load the values on in_op(2:0) into op_1 and the contents of op_1 into op_2

corr. The correction flip-flop **corr** is used to indicate that a ten's complement of the contents of **ACC_R** is to be taken and the **sign** of the result for **ACC_R** is to be changed. **corr** hold its present contents, can be loaded from $\overline{\textbf{c\_msb}}$ or can be reset to 0 or can be set to 1, all synchronously. The following table describes the operation of **corr**:

corr		
Op Name	**corr_c(1:0)**	**Operation**
hold	0 0	Hold the contents of corr
reset	0 1	Reset **corr** to 0
set	1 0	Set **corr** to 1
load	1 1	Load $\overline{\textbf{c\_msb}}$ into **corr**

sign. The **sign** flip-flop holds the sign of the current result. sign is actually a two-state sequential circuit and functions as described in the following state table:

sign				
sub	**corr**	**sign(t)**	**sign(t + 1)**	**Comments**
0	0	0	0	The sign changes if the operation performed in the datapath is subtraction, sub = 1, and the carry out of the most significant bit, c_msb = 0, i. e., corr = 1. Note that the calculator operation causing this may be either addition or subtraction.
0	0	1	1	
0	1	0	0	
0	1	1	1	
1	0	0	0	
1	0	1	1	
1	1	0	1	
1	1	1	0	

op_tran. The operation translation logic, **op_tran**, converts the calculator operation to be performed into the datapath operation to be performed. **op_tran** is defined by the following truth table:

op_tran

Operation	op_2(2)	op2(1)	op2(0)	sign	corr	sub	comp_a
Add	**0**	0	0	1	0	**1**	**0**
Sub	**0**	0	1	0	0	**1**	**0**
Clear	**1**	0	0	X	0	0	0
Equal	**0**	1	0	X	0	0	0
10's comp A	**X**	X	X	X	1	0	1
	All other combinations					X	X

fsm. The state machine, fsm, provides all of the sequencing and control signals for both the datapath and the control itself. The sequencing is relatively simple; however, there are many inputs and many outputs to be generated. As a consequence, we will separate the sequencing of the states, i. e., the next state logic, from the generation of the control signal values, the output logic. The state flip-flops and sequencing logic will be in component **st_seq** and the output logic will be in component **output**.

st_seq. The sequencing of the state machine is represented by the state diagram in Figure 6. Only the inputs that affect the state sequence are shown. We will deal with the output control signals later. The typical actions that take place in each of the states are as follows:

S0. **S0** is the **Reset** state. Also, **S0** is also the state that is reached by a **Clear** operation. In this state, any operations entered will be ignored. Until a digit is enter (**en_dig** = 0), no changes in calculator state occur. When **en_dig** becomes 1, the first digit is entered into **ENT_R**.

S1. This is the **Digit Entry** state. While no operand is entered (**en_op** = 0), additional digits may be entered in this state and the state remains unchanged. When an operand is entered (**en_op** = 1), the **op_queue** is advanced. This places the prior operation in **op_2** so it is ready for execution. If **S1** was entered from **S0**, the **op_queue** would have been reset to all zeros, so the operation advanced is 000 (**Add**). If **S1** is entered from **S7**, the operation to be executed will be the last one entered while in **S7**.

S2. This is the **Operation Setup** state. In addition, it is one of two states in which a test is made to see if a **Clear** has been entered into **op_1** in **op_queue**. If a **Clear** appears in **op_1** (**op_1(2)** = 1), then all parts including the state_register in the control are given (synchronous) reset control codes, clearing all flip-flop states and going to state **S0**. If a digit is entered immediately after =, **ACC_R** and **sign** are cleared. Otherwise, **carry** in the datapath is initialized to 0 for **add** and 1 for **sub** or **10's comp** in preparation for the operation to begin.

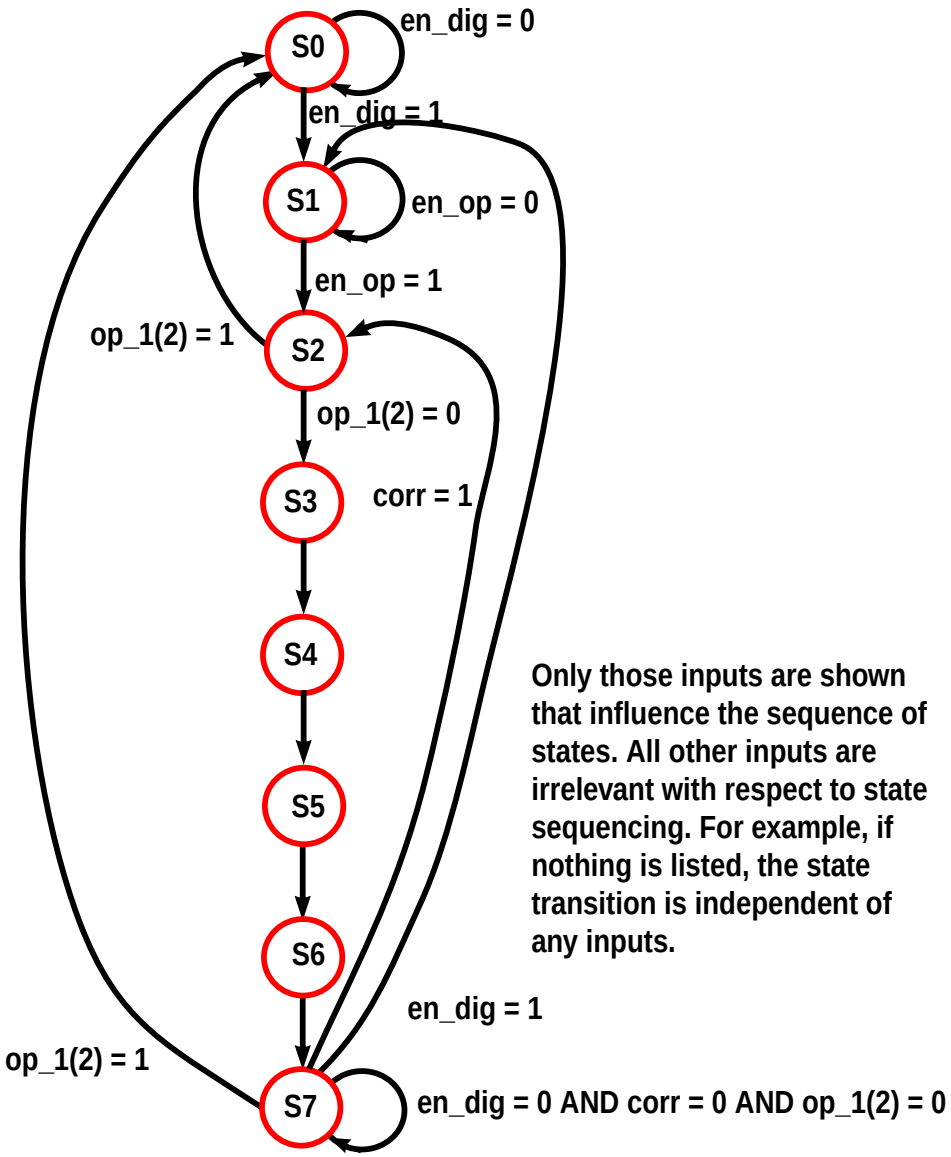
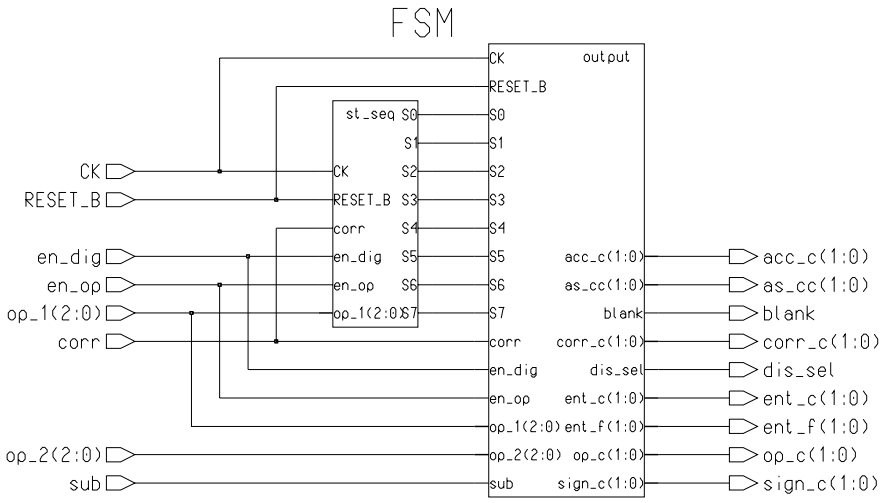


Figure 6: fsm – State Sequencing part

S3 – S6. These are the four **Operation Execute** states. In each state, one of the digit positions of the operands are processed. In **S3**, digit 0 is processed and, in **S4**, digit 1 is processed, etc. The operations performed are **add**, **sub**, and **10's comp**. The **10's comp** is used for the correction for subtraction and is controlled by flip-flop **corr**. If **corr** = 0, then **add** or **sub** occur. If **corr** = 1, then the **10's comp** of **ACC_R** is taken. **corr** is set during **sub** if the carry out of digit 3 is a 0 (**c_msb** = 0 in state **S6**). The setting (and resetting) of **corr** is controlled in state **S6**.



S7. S7 is the **Operation Update** state. In addition, it is one of two states in which a test is made to see if a **Clear** has been entered into **op_1** in **op_queue**. If a **Clear** has occurred (**op_1(2)** = 1), then all parts including the state_register in the control are given (synchronous) reset control codes, clearing all flip-flops and going to state S0.

Also, if a subtraction has just been executed and, as a result, **corr** was set in state **S6**, then the next state will be **S2** to perform the correction. If **corr** was set at the end of a **sub**, then it is necessary to bypass the advance of **op_queue** in state **S1** since the **10's comp** correction is performed instead of a calculator operation. Thus the next state is **S2**.

Otherwise, in state **S7**, as long as a digit is not entered (**en_dig** = 0), operations may be entered into **op_queue**. Only the last operation entered before (**en_dig** = 1) will be executed. When **en_dig** = 1, the first digit is entered into **ENT_R** and the next state is **S1** in preparation for processing additional digits.

state_reg. The sequencing logic for this state diagram will use a "one flipflop per state" state assignment in a module called **state_reg**. For this assignment and the corresponding design approach, see sections 8-2 and 8-4 of Mano and Kime. This register consists of eight D flip-flops, one for each of the states. The next state inputs to **state_reg** are to be called **NS0** through **NS7** and the state outputs are to be called **S0** through **S7**. For **RESET_B**, the flip-flop for **S0** is to be initialized to 1, so **RESET_B** will be attached to its **Preset** input. **RESET_B** resets all other flip-flops to 0.

st_seq. This is the combinational logic module that implements the next state functions **NS0** through **NS7**. It can be designed from the state diagram in Figure 6. Its inputs are shown in on the state diagram and in Figure 7. Its outputs are lines representing each of the states, **S0** through **S7**.

output. This is the combinational logic module that implements all of the control signal outputs from **fsm**. Its inputs and outputs are shown in Figure 7. **output** produces the control signals from the **fsm** can be designed based on the above state descriptions plus

additional reasoning as to how the calculator works. Formation of an output table which can be entered into CAFE as a truth table or transformed into equations is the easiest way to approach the development. Such a output table listing all states, inputs to **fsm**, and control signal outputs from **fsm** is given as Table 1.The tables in this writeup which specify the control variable values and corresponding actions were very useful in determining these values. Note the use of don't cares in many places. This is important for permitting simplification, but care must be taken not to specify a don't care erroneously. Quickly going over the first several rows of the table:

TABLE 1. fsm Output Specification

Row No.	INPUTS							OUTPUTS								
	State	corr	en_dig	en_op	op_1(2:0)	op_2(2:0)	sub	acc_c(1:0)	as_cc(1:0)	blank	corr_c(1:0)	dis_sel	ent_c(1:0)	ent_f(1:0)	op_c(1:0)	sign_c(1:0)
1	S0	X	0	X	X	X	X	00	00	1	00	X	00	00	00	00
2	S0	X	1	X	X	X	X	00	00	1	00	X	11	10	00	00
3	S1	X	0	0	X	X	X	00	00	0	00	0	00	00	00	00
4	S1	X	1	X	X	X	X	00	00	0	00	0	10	10	00	00
5	S1	X	X	1	X	X	X	00	00	0	00	0	00	00	10	00
6	S2	X	X	X	(2)=1	X	X	11	01	1	01	X	11	11	01	01
7	S2	0	X	X	(2)=0	000	X	00	01	1	00	X	00	00	00	00
8	S2	0	X	X	(2)=0	X	1	00	10	1	00	X	00	00	00	00
9	S2	0	X	X	(1)=1	X	X	11	01	1	X	X	X	X	X	X
10	S2	1	X	X	(2)=0	X	X	00	10	1	00	X	00	00	00	10
11	S3	X	X	X	X	X	X	01	11	1	00	X	01	01	00	00
12	S4	X	X	X	X	X	X	01	11	1	00	X	01	01	00	00
13	S5	X	X	X	X	X	X	01	11	1	00	X	01	01	00	00
14	S6	X	X	X	X	X	X	01	11	1	00	X	01	01	00	00
15	S6	0	X	X	X	X	1	01	X	X	11	X	X	X	X	X
16	S6	1	X	X	X	X	1	01	11	1	01	X	X	X	X	X
17	S7	X	X	X	(2)=1	X	X	11	01	0	00	1	11	11	01	01
18	S7	X	0	0	(2)=0	X	X	00	00	0	00	1	00	00	00	00
19	S7	X	0	1	(2)=0	X	X	00	00	0	00	1	00	00	10	00
20	S7	X	1	X	(2)=0	X	X	00	00	0	00	1	11	10	00	00

Row 1. In this row, in state **S0**, operations may be entered, but with **en_dig** = 0, nothing is happening. So all control fields are set to **hold**.

Row 2. In state **S0**, with **en_dig** = 1, **ent_c** is reset (11) and **ent_f** is **sl** (shift left) (10) to load the first digit into **ENT_R** and clear the rest of the digits.

This completes all of the cases for state S0. Note that it is very important to consider all distinct input cases that must be defined for each of the states.

Row 3. In state **S1**, with both **en_dig** = 0 and **en_op** = 0, nothing happens, so all control fields are **hold**.

Row 4. In state **S1**, with **en_dig** = 1 and **en_op** = 0, a new digit is entered into **ENT_R** by **sl**, so both **ent_c** and **ent_f** are 10.

Row 5. In state **S1**, with **en_op** = 1, digit entry is completed since an operation has now been entered. The operation code goes into **op_1**, and the code in **op_1** is transferred to **op_2**. This requires that **op_c** be 10, the code for **advance**.

Similar reasoning was used to develop the entries for the remaining rows of the table.

The modules **state_reg**, **st_seq**, and **output** can be interconnected to form the **fsm**. The interconnection is shown in Figure 7. **state reg** is inside of **st_seq**.

CONTROL DESIGN

This section gives a step-by-step process to design the **core** module. Some initial guidelines:

1. Be sure to name all inputs and outputs and modules exactly as given.
2. Use buses (bundles) for anything that is shown with bit numbers in ( ). Do use buses if ( ) not given.
3. Remember that you must label a bus before you connect the wires to it.
4. A constant 0 can be obtained by connecting to the **gen_lib** component **ground**.
5. A constant 1 can be obtained by connecting to the **gen_lib** component **vcc**.

Proceed in the design and validation of the control hierarchy and core as follows.

The following cells are specified in the **Control Specification** section.

The **op_queue** is to be design as a cell, two 3-bit registers implemented with the cells, and the overall queue implemented with the two registers.

op_queue_cell. **op_queue_cell** consists of a single flip-flop driven by some combinational logic. It has two active operations, **reset** (synchronous) (**op_c** = 01) and **advance (op_c = 10)**, which is a "load," as well as the inactive operation **hold (op_c** = 00) as specified by the table given earlier for **op_queue**. Design **op_queue_cell** using **da**; create a symbol. The names to be used are: **CK**, **RESET_B**, **op_c(1:0)**, **op_bit**, and **op_out**. **op_c(1:0)** is a 2-bit bundle, **op_bit** is the data input to be loaded for advance and **op_out** is the flip-flop output. Using **quicksim**, manually apply some inputs to the cell to be sure that it functions correctly for each of the control combinations and data input values. Print out your schematic when finished.

Submit: printed schematic of **op_queue_cell**.

op_queue_3_cell. Design **op_queue_3_cell** using **da**; create a symbol. Interconnect three copies of **op_queue_cell**, connecting together the corresponding inputs and outputs. The three **op_bit** lines become bus **in_op(2:0)** and the three **op_out** lines become

bus **out_op(2:0)**. Manually apply some inputs in **quicksim** to **op_queue_3_cell** to be sure it functions correctly. Print out your schematic when finished.

Submit: printed schematic of **op_queue_3_cell**.

op_queue. Design **op_queue** using **da** from two **op_queue_3_cells** with the **out_op(2:0)** output from the first register (**op_1**) connected to **in_op(2:0)** input on the second cell (**op_2**). Connect input port **in_op(2:0)** to **in_op(2:0)** on **op_1** and output port **op_1(2:0)** on **out_op(2:0)** on **op_1(2:0)**. Connect output port **op_2(2:0)** on **out_op(2:0)** on **op_2**. Connect together **CK**, **RESET_B**, and **op_c(1:0)**. Create a symbol. Manually apply some inputs in **quicksim** to **op_queue** to be sure it is logically correct. Print out your schematic when finished.

Submit: printed schematic of **carry_gen** and a print of the trace window contents for your test of **op_queue** with comments to show that **carry_gen** is working correctly.

The next parts of the control are three blocks of logic that it are inconvenient to include in the fsm because they would increase the number of states substantially or increase the complexity.

sign. Design **sign** by using **da** using a J-K flip-flop with **CK** as the CLK and **RESET_B** as the CLR. Use the table given in the **Control Specifications** to define the connection of **sign_c(1:0)** to the flip-flop. Create a symbol. Manually apply some inputs in **quicksim** to **sign** to be sure it is logically correct. Print out your schematic when finished.

Submit: printed schematic of **sign.**

corr. Design **corr** by using **da** using a flip-flop and combinational logic. Use the table given in the **Control Specifications** to define the logic. Minimize your logic. Create a symbol. Manually apply some inputs in **quicksim** to **corr** to be sure it is logically correct. Print out your schematic when finished.

Submit: printed schematic of **corr** and provide a print of the trace window contents for your test of **corr** with comments to show that **corr** is working correctly.

op_tran. Design combinational block **op_tran** by using **da**; create a symbol. Use the table given in the **Control Specifications** to define the logic and inputs **op_2(2:0)** and **corr** and output **sub** and **comp_a**. Minimize your logic. Manually apply all input combinations to **op_tran** and use list window output in **quicksim** to be sure it is logically correct. Print out your schematic and the list window when finished; comment the list window content to indicate that **op_tran** is working correctly.

Submit: printed schematic of **corr** and provide a print of the list window for your test of **corr**.

Next is the **fsm** which will be designed in three parts and then integrated. First is the state register. This register will be included in the state sequencing logic (**st_seq**) which is the second part. Finally is the **output** logic which is driven by the inputs to **fsm** and the state lines from the state sequencing logic. The inputs and outputs of **st_seq** and **output** are available in Figure 7 which shows their interconnection to form the **fsm**.

state_reg. This register consists of eight D flip-flops with **CK** as the CLK and **RESET_B** as the CLR. The outputs of the flip-flops are the register outputs with names

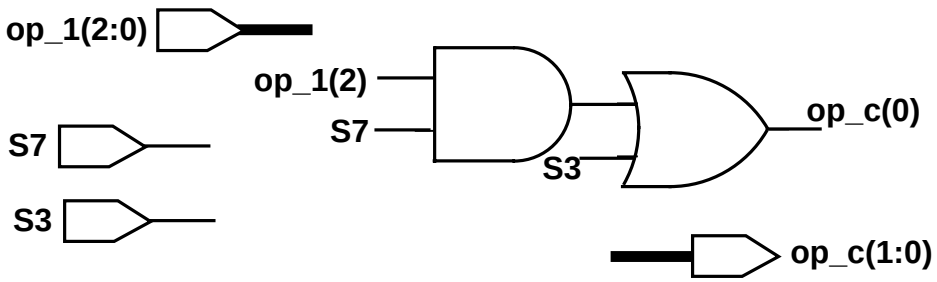


Figure 8: Illustration of connection by naming

S0, **S1**, ..., **S7** and the inputs to the flip-flops are correspondingly named **NS0**, **NS1**, ..., **NS7**. Design **state_reg** by using **da**; create a symbol. Print out your schematic when finished.

Submit: printed schematic of **state_reg**.

st_seq. The state sequencing logic contains the state register and combinational logic implementing **NS0**, **NS1**, ..., **NS7**. In those cases where **NSj** = **Si**, use a **buf** from **gen_lib** between **Si** and **NSj**. To obtain input **op_1(2)**, attach bus **op_1(2:0)** to **st_reg** and internally rip off **op_1(2)**. Design the combinational logic using techniques from section 8-2 and 8-4 of Mano and Kime. The **st_seq** state diagram is like an ASM and a "one flip-flop per state" state assignment is being used. Enter design into **da** and create a symbol. Manually apply all input combinations to **st_reg** (use list window output in **quicksim)** to traverse every arc in the state diagram for each input combination specifically specified on the arc. Print out your schematic and the list window when finished.

Submit: printed schematic of **st_seq** and provide a print of list window contents for your test of **st_reg**. Comment to show that the content of the list window is consistent with the state diagram.

output. output consists of combinational logic with S0, S1, and S7 and various inputs as inputs and many control outputs. The inputs and outputs are shown in Figure 7. The logic can be designed from Table 1 given earlier. Equations can be developed for each of the output variables equal to 1 and then reduced manually or by using CAFE. Due to the complexity of this block, entry into da should use a different approach. In Mentor Graphics tools, if two lines at a given level of hierarchy have the same name, they are automatically connected! Thus, you need not draw lines from place-to-place to make connections. You can simply attach a short net to every input or output in the circuit and whenever you assign two points the same name, they will be connected. In particular, with an input labeled **op_1(2:0)**, you can label a particular gate input **op_1(2)** to get just a since signal from this input bundle. This is illustrated in Figure 8. The logic is in small clusters and bits of nets (including bundles) are attached to each port and unconnected gate inputs and outputs. Naming is then use to implement connections and rippers. It is suggested that a cluster be used for each output. Design combinational block **output** by using **da**; create a symbol. Manually apply selected input combinations based on specified entries in Table 1 to **output** and use list window output in **quicksim** to validate its logical correctness somewhat. Print out your schematic and the list window when finished.

Submit: printed schematic of **output** and provide a print of list window contents for your test of **output**. Comment to show that the list window content is consistent with Table 1.

fsm. Enter **fsm** in Figure 7 into **da**; create a symbol. Print your **fsm** schematic when finished and carefully check it over including net names.

Submit: printed schematic of **fsm.**

control. Enter **control** in Figure 5 into **da**; create a symbol. Print your schematic **control** when finished and carefully check it over including net names.

Submit: printed schematic of **control.**

calc. Enter the calculator core **calc** in Figure 3 into **da**. We will provide a **quicksim** force file for testing **calc** on the web site. Use this force file to do the test. By watching the core inputs, and outputs **sign**, and **dis3** through **dis0** while blank = 0, it will appear as if you were looking at the calculator display. Print out your schematic and traces when finished.

Submit: printed schematic of **calc** and the trace output from **quicksim**. Comment the trace output to show that the calculations are correct.

SUBMISSION

Staple together all of the required submissions exactly in the order given above and submit them as directed by your instructor. Please do not submit to mailboxes. Submission instructions for other than in class should be obtained from your instructor.

TEAM EFFORT REPORT

Submit: This page as the **last** page of your project report; the **next to last** page is the **Team Effort Report** for Part 1. Each **row** of the table containing the task contributions must sum to 100%.

Team Member Names:		
op_queue hierarchy: Entry	%	%
op_queue hierarchy: Validation	%	%
op_queue hierarchy: Debug	%	%
sign: Entry	%	%
sign: Validation	%	%
sign: Debug	%	%
corr: Entry	%	%
corr: Validation	%	%
corr: Debug	%	%
op_tran: Design/Entry	%	%
op_tran: Validation	%	%
op_tran: Debug	%	%
state_reg: Design/Entry	%	%
state_reg: Validation	%	%
state_reg: Debug	%	%
st_seq: Design/Entry	%	%
st_seq: Validation	%	%
st_seq: Debug	%	%
output: Design/Entry	%	%
output: Validation	%	%
output: Debug	%	%
fsm: Design/Entry	%	%
control: Design/Entry	%	%
calc: Design/Entry	%	%
calc: Validation	%	%
calc: Debug	%	%
Other:	%	%
Other:	%	%
Other:	%	%

Comments: