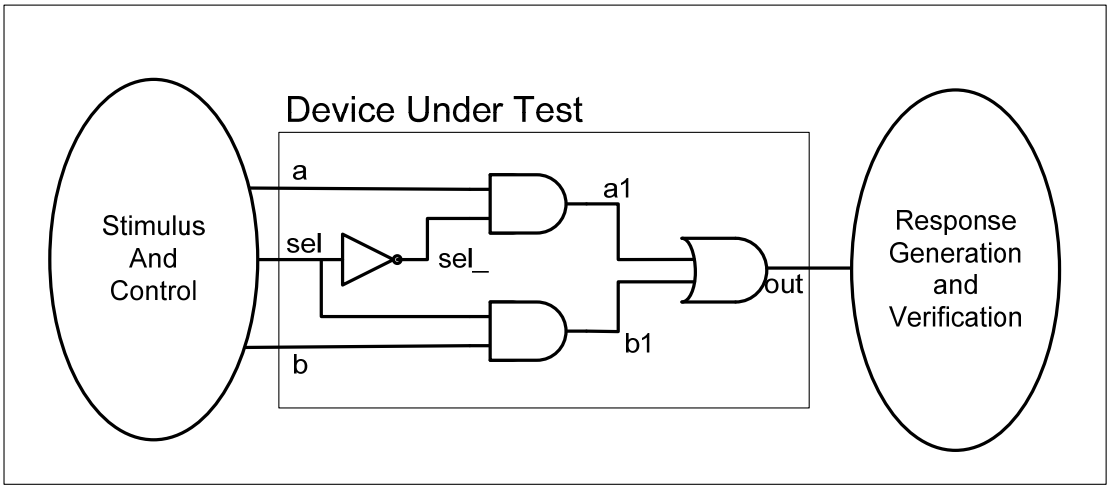
Lab1 : 2-1 MUX

- Please design a 2-1MUX
- Specifications
  - Module name : mux
  - Input pins : a, b, sel
  - Output pins : out
  - Function :

Test Fixture



1. Change to **Lab1** directory. It contains the *mux.v* and *mux_text.v* files. use this command : cd Lab1

2. You can edit the ***mux*** module (*mux.v*) accord to the above DUT schematic . The content of *mux.v* is as follows:

```
`define  dly_and  1
`define  dly_or  2
module MUX (out,a,b,sel);
// Port declarations
output      out;
input       a,b,sel;
// The netlist
     not                  not1(sel_, sel);
     and      #`dly_and   and1(a1, a, sel_);
     and      #`dly_and   and2(b1, b, sel);
     or       #`dly_or    or1(out, a1, b1);
endmodule
```

3. Edit the test bench (*mux_test.v*)
   The signal declarations, model instantiation, and response generation are written for you. The register inputs to the ***mux*** are initialized and the simulation with finish at time 40. add vectors to test ***mux*** according to the following table :

time	a	b	sel
10	0	0	0
20	1	0	0
30	0	0	1
40	0	1	1

4. Setup the waveform display.

```
initial       begin
    $dumpfile("mux.vcd");      // The VCD Database
    $dumpvars;
    $fsdbDumpfile("mux.fsdb"); // The FSDB Database
    $fsdbDumpvars;
    $shm_open("mux.shm");    // The SHM Database
    $shm_probe("AC");
end
```

5. Simulate the design, enter

```
verilog   mux_test.v   mux.v
```

   if you using NC-Verilog, enter

```
ncverilog   mux_test.v   mux.v   +access+r
```

   Note : In this and all subsequent labs, the command *verilog* is used to invoke the simulator. To simulate with *NCVerilog*, the *+access+r* option allows you to view signal in the wave tool.

6. without `define used, the simulation results will be similar to this :

```
0   a = x,   b = x,   sel = x,   out = x
10  a = 0,   b = 0,   sel = 0,   out = 0
20  a = 1,   b = 0,   sel = 0,   out = 1
30  a = 0,   b = 0,   sel = 1,   out = 0
40  a = 0,   b = 1,   sel = 1,   out = 1
```

Wave Tool and Waveform Database

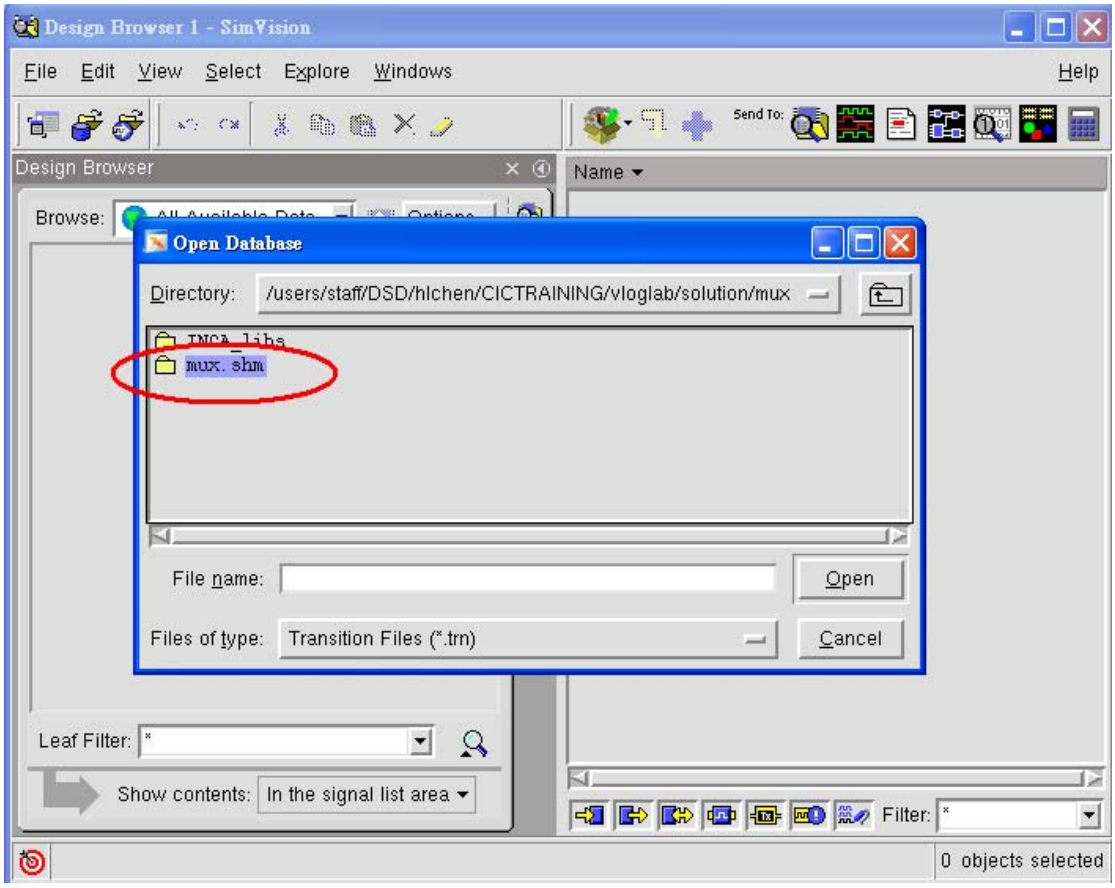
◆ cadence simvision waveform viewer

1. In the command shell, open the waveform tool from the same directory where you started the simulation.
   To invoke *simvision*, enter :

   simvision &

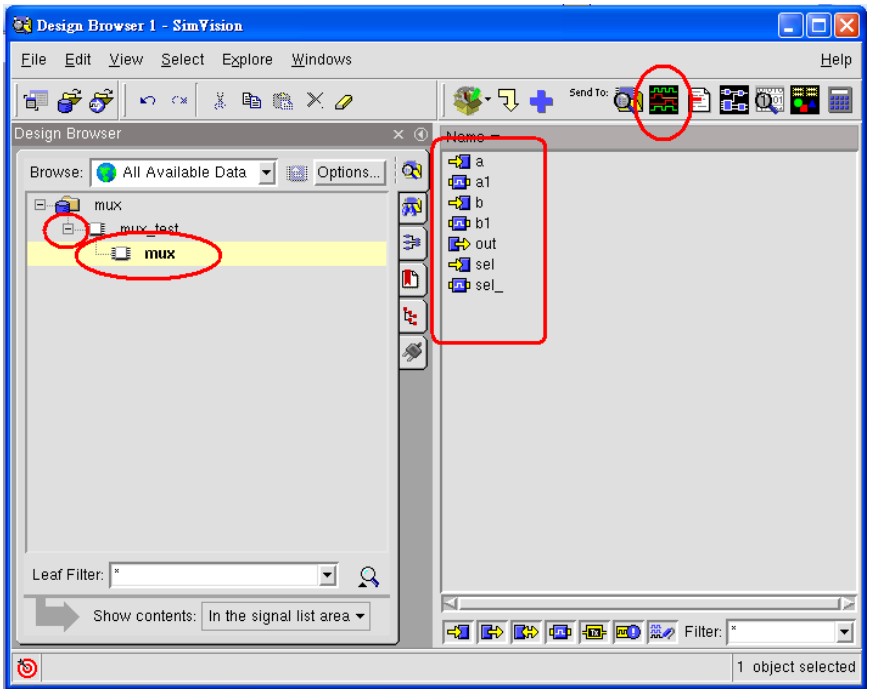
   or supply the database name as an argument.

   simvision file.shm &
2. In the waveform window, select **File – Open Database** file. The file browser appears.



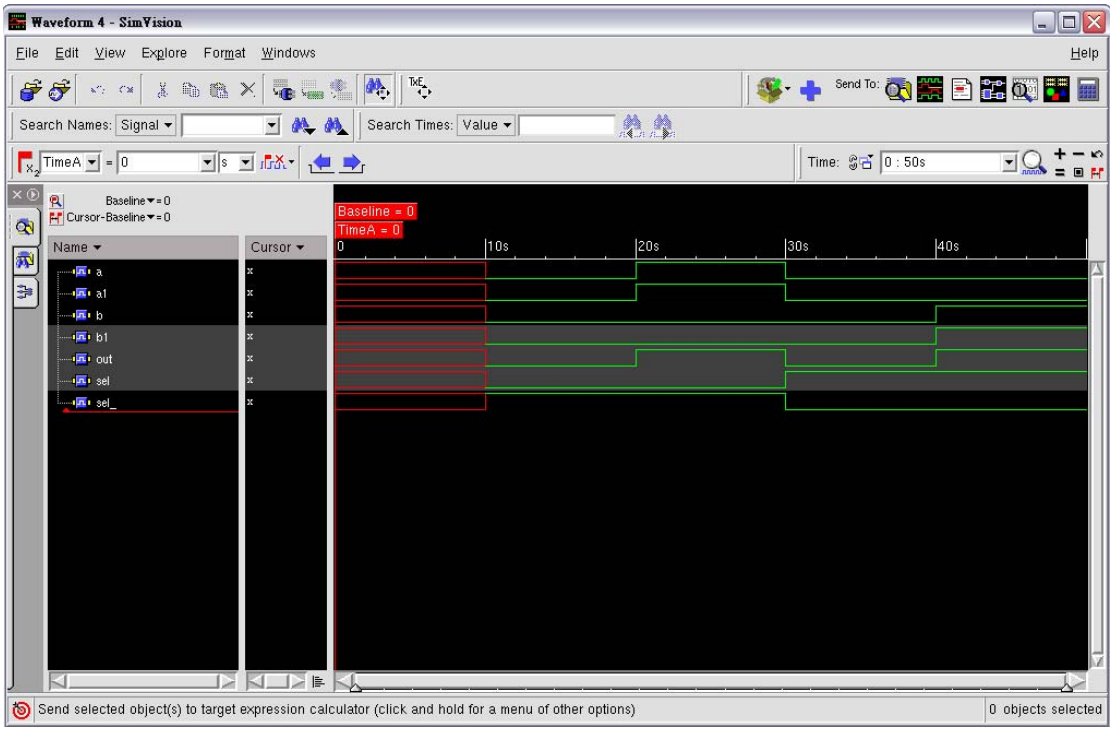
   Select the *file.shm* database, click **Open**
   Then select the *file.trn*, click **Open**

3. From the *simvision Design Browser*, click the launch of the ***mux_text*** module to find the ***mux*** module signal view.

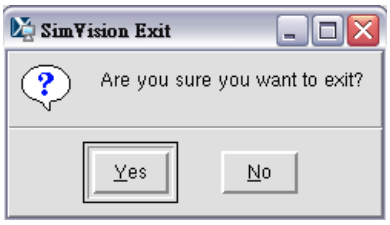


4. select the ***mux*** module instance, you can find all the pins of the module in the right browser. Shown as over.

5. click the **Waveform** icon, the waveform window appears .



6. From the *simvision* menu, select **File - Exit simvision**.
   A pop-up window appears to verify your intentions.

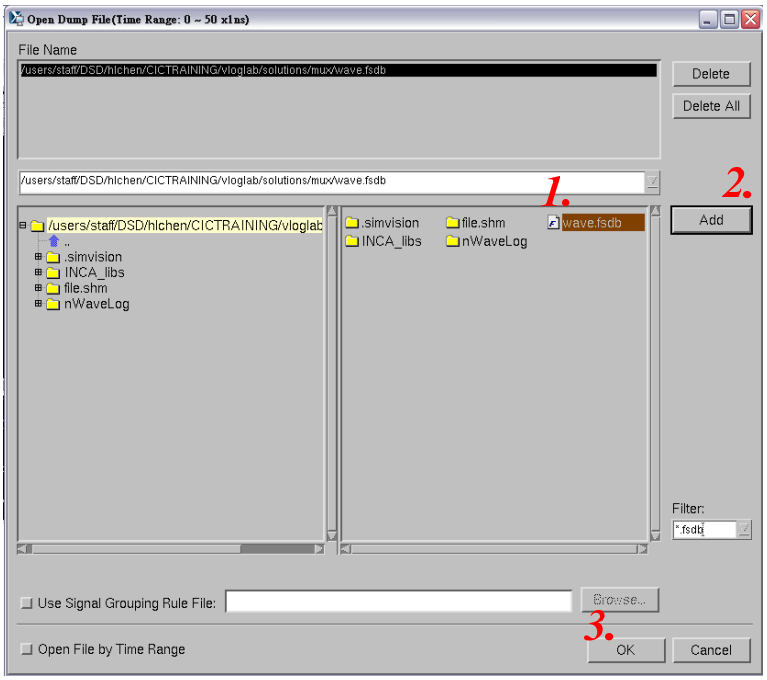


7. Click **Yes**.
   The *simvision* window closes.

◆ Verdi – nWave

1. In the command shell, open the waveform tool from the same directory where you started the simulation.
   To invoke *nWave*, enter :
   nWave &

2. In the waveform window, select **File – Open** . The file browser appears.



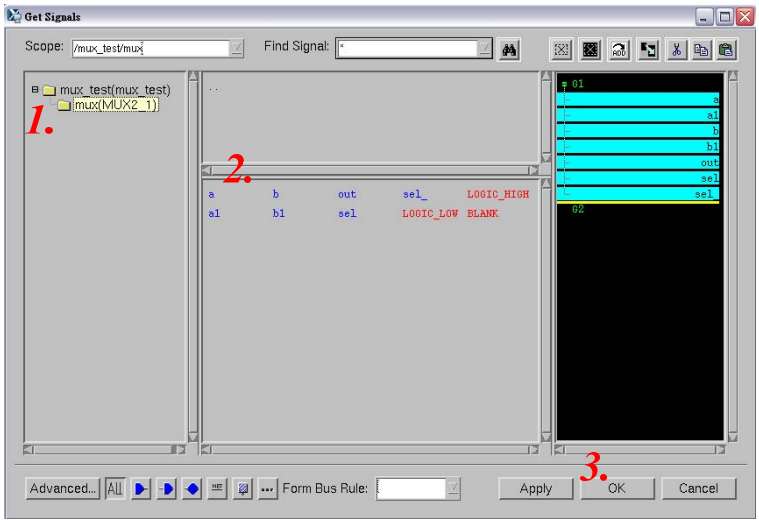
   i. Select the *wave.fsdb* database,
   ii. click Add
   iii. click OK

3. From the *nWave window*, select **signal – get signal**.
   The Get signal window appears.

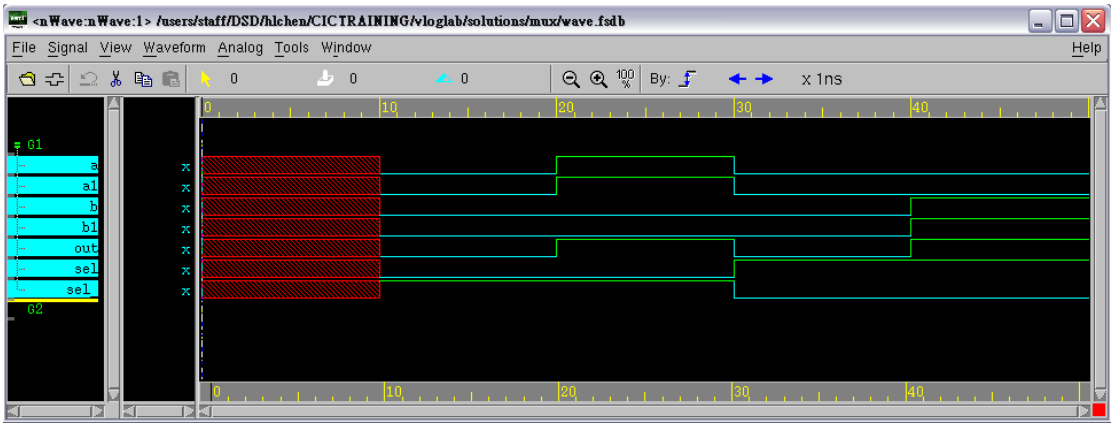
i. You can select ***mux_test*** or ***mux*** to find the I/O signal of the module.

ii. Select the signal to scope.

iii. Click OK



4. Then the waveform will be shown in the *nWave browser*.



5. From the *nWave* menu, select **File – Exit**,
A pop-up window appears to verify your intentions.

6. Click **Yes**.
The *nWave* window closes.

End of Lab1

Lab2 : Full Adder Module Design

- Please design a Full-Adder
- Specifications
  - Module name : fa
  - Input pins : a, b, ci
  - Output pins : sum, cout
  - Function : { cout, sum } = a + b + ci;
  - Truth Table :

a	b	ci	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- **Solution 1 :**

1. Change directory to ***Lab2***. It contains the *fa.v* and *fa_test.v*

```
cd  Lab2
```

2. start modeling the full-adder in the file called *fa.v* . Using the truth table on the over, instantiate components from the built-in primitives in Verilog.

```
module fa(a, b, ci, sum. cout);
    input    a, b, ci;
    output   sum, cout;
    and   g1(……);
    not    g2(……);
        ……
endmodule
```

3. simulate with *verilog-XL* , enter :

```
verilog  fa_test.v  fa.v
```

If you using *NC-Verilog*, enter :

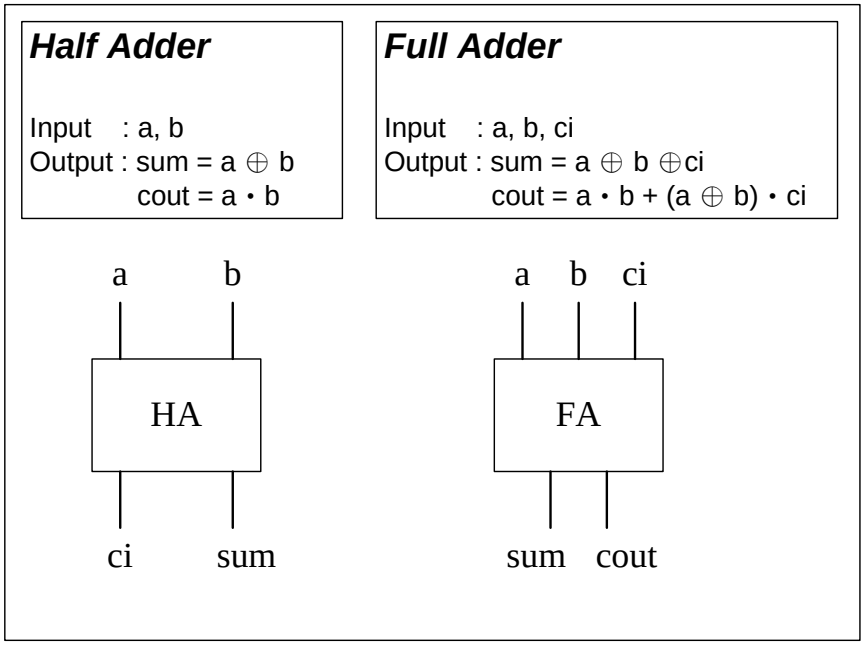
```
ncverilog  fa_test.v  fa.v  +access+r
```

4. If the function is correct, the simulation results will be similar to this :

```
**********************
function test pass !!
**********************
```

◆ **Solution 2:**

Half-Adder and Full-Adder



1. Change directory to ***Lab2***. It contains the *fa.v*, *ha.v* and *fa_test.v*

```
cd  Lab2
```

2. start modeling the full-adder in the file called *fa.v*. Use the following module header and port interface for the full-adder module:

```
module  fa(a, b, ci, sum, cout);
    output  sum, cout;
    input   a, b, ci;
    .....
endmodule
```

The full-adder module can be composed of two Half-adder. Modeling the Half-adder (*ha.v*), using the logic solution on the over of this lab, instantiate components from the built-in primitives in *Verilog*, and has following interface :

```
module    ha(a, b, sum, cout);
     output    sum, cout;
     input      a, b;
     ……
endmodule
```

3. create a verilog control file named *run.f* . this control file should specify the design and testbench file, and contain the appropriate command-line options to specify. After edit the *run.f* file, simulate with the Verilog control file :

```
verilog    -f    run.f
```

4. If the function is correct, the simulation results will be similar to this :

```
**********************
function test pass !!
**********************
```

End of Lab2

Lab2b : 4-bits Adder/Subtractor Design

- Please design a 4-bits Adder/Subtractor Module
- Specifications
  - Module name : ADDER
  - Input pins : a[3:0], b[3:0], add
  - Output pins : sum[3:0], overflow
  - Function : { cout, sum } = a + b + ci;
  - Truth Table :

a	b	ci	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- **Solution 1 :**

1. Change directory to ***Lab2***. It contains the *fa.v* and *fa_test.v*

```
cd  Lab2
```

2. start modeling the full-adder in the file called *fa.v* . Using the truth table on the over, instantiate components from the built-in primitives in Verilog.

```
module fa(a, b, ci, sum. cout);
    input    a, b, ci;
    output   sum, cout;
    and   g1(……);
    not    g2(……);
        ……
endmodule
```

3. simulate with *verilog-XL* , enter :

```
verilog  fa_test.v  fa.v
```

If you using *NC-Verilog*, enter :

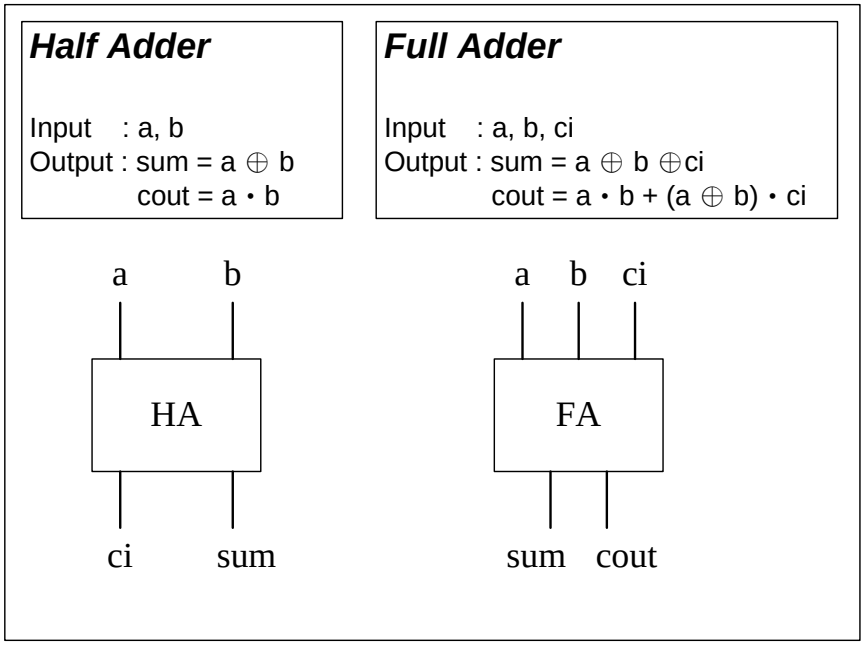
```
ncverilog  fa_test.v  fa.v  +access+r
```

4. If the function is correct, the simulation results will be similar to this :

```
**********************
function test pass !!
**********************
```

◆ **Solution 2:**

Half-Adder and Full-Adder



1. Change directory to ***Lab2***. It contains the *fa.v*, *ha.v* and *fa_test.v*

```
cd  Lab2
```

2. start modeling the full-adder in the file called *fa.v*. Use the following module header and port interface for the full-adder module:

```
module  fa(a, b, ci, sum, cout);
    output  sum, cout;
    input   a, b, ci;
    .....
endmodule
```

The full-adder module can be composed of two Half-adder. Modeling the Half-adder (*ha.v*), using the logic solution on the over of this lab, instantiate components from the built-in primitives in *Verilog*, and has following interface :

```
module    ha(a, b, sum, cout);
    output    sum, cout;
    input      a, b;
    ……
endmodule
```

3. create a verilog control file named *run.f* . this control file should specify the design and testbench file, and contain the appropriate command-line options to specify. After edit the *run.f* file, simulate with the Verilog control file :

```
verilog    -f    run.f
```

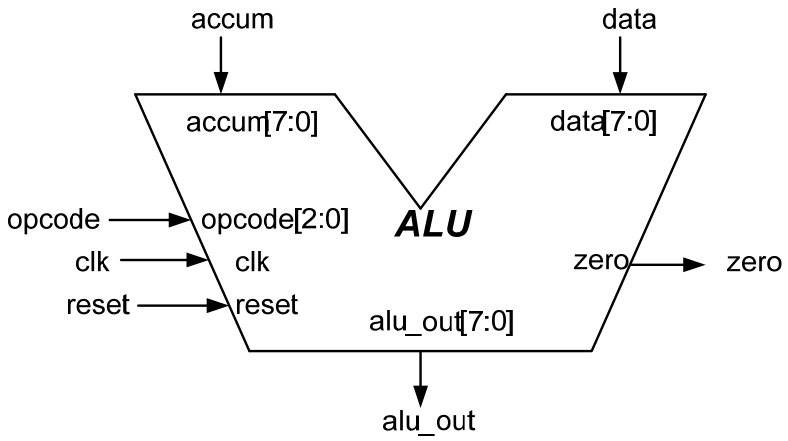
4. If the function is correct, the simulation results will be similar to this :

```
**********************
function test pass !!
**********************
```

End of Lab2

Lab3 : ALU

- Please modeling an Arithmetic Logic Unit (ALU)
- Specifications
  - Module name : alu
  - Input pins : accum[7:0], data[7:0], opcode, clk, reset
  - Output pins : zero, alu_out[7:0]
  - Function :



1. Change the directory to ***Lab3.*** This directory contains the stimulus file *alu_test.v* and alu module file *alu.v* **.**

2. Model the ALU in the *alu.v* . Model port interface according to the symbol view on the over. Model the functionality according to the following specifications.
   i. All inputs and outputs(exclude "*zero*" signal) are synchronized at clock rising edge .
   ii. It is a synchronous-reset architecture. *alu_out* become 0 when the reset equal 1.
   iii. *accum*, *data* and *alu_out* are using 2's complement expression.
   iv. The *zero* bit become 1 when the *accum* equal 0, and is 0 otherwise.
   v. The *alu_out* becomes 0 when *opcode* is X(unknow).

vi. Its value is determined when the ALU decode the 3-bits *opcode* and performs the appropriate operation as listed below.

opcode	ALU operation	
000	Pass accumulator	
001	accumulator + data	(addition)
010	accumulator - data	(subtraction)
011	accumulator AND data	(bit-wise AND)
100	accumulator XOR data	(bit-wise XOR)
101	ABS(accumulator)	(absolute value)
110	NEG(accumulator)	(negate value)
111	Pass data	

p.s. opcode 為 absolute value 時，使用 accum[7]當作 signed bit

3. Test your ALU model using the *alu_test.v* file
simulate with *verilog-XL* , enter :
verilog alu_test.v alu.v
If you using *NC-Verilog*, enter :
ncverilog alu_test.v alu.v +access+r

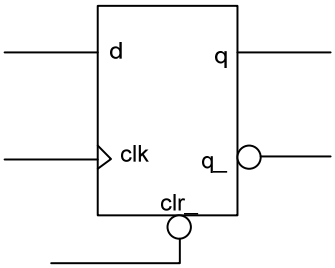
4. Check the result.

End of Lab3

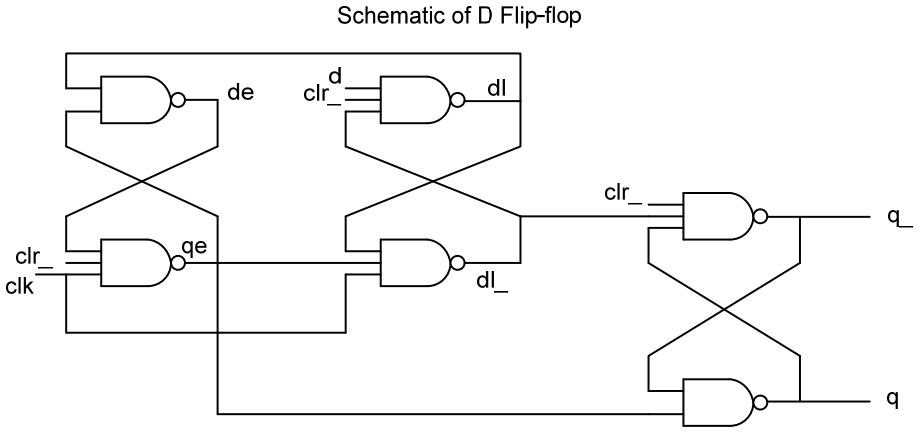
Lab4 : Modeling Delay

- Use path delays and timing checks to improve the accuracy of a model.
- Specifications
  - Module name : dffr_b
  - Input pins : clr_, clk, d
  - Output pins : q, q_
  - Symbol and Schematic :

Symbol of D Flip-flop



Schematic of D Flip-flop



1. Change the directory to ***Lab4.*** This directory contains the stimulus file *test.v* and design module file ***dffr_b.v .***

2. Review the *test.v* file provided in this directory. Notice the use of the signal *flag*. The signal *flag* is a notifier, declared and set by timing violations in ***dff_b***.

3. Edit and modify the D Flip-flop model with the timing information shown below.

   i. Add a specify block to the model with the timing information

shown below.

ii. In your setup and hold timing checks, include a notifier. Be sure to name it *flag*, as it is referred to hierarchically from *test.v*. Because it is assigned a value by the timing violations, you must declare it to be of type **reg**.

Pin-to-Pin Path Delays		***min***	***typ***	***max***	***min***	***typ***	***max***
inputs	***outputs***		***rise***			***fall***	
clr_	q, q_	3	3	3	3	3	3
clk	q	2	3	5	4	5	6
clk	q_	2	4	5	3	5	6

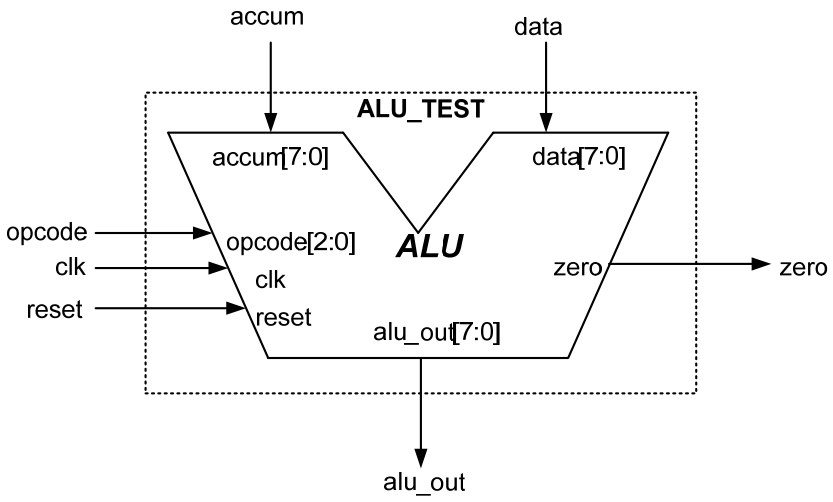
Timing Constraints	min	typ	max
Tsu (setup)	3	5	6
Th (hold)	2	3	6

4. Simulate the design with a command-line option.
   Did you get any timing violation in min, typ or max delay mode ?
   By default, were minimum, typical, or maximum delays used?

End of Lab4

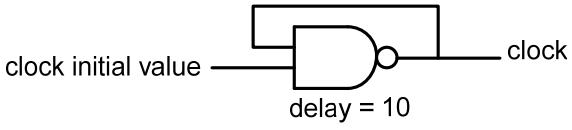
Lab5 : Testbench of ALU

- Please modeling an Arithmetic Logic Unit (ALU)
- Specifications
  - Module name : alu_test
  - Interface with ALU : accum[7:0], data[7:0], opcode, clk, reset, zero, alu_out
  - Function :



1. Change the directory to ***Lab5.*** This directory contains the stimulus file *alu_test.v* and function module file *alu.v* **.**you need to edit the *alu_test.v* to finish the testbench.

2. Edit the testbench in the *alu_test.v* .
   i. Instantiate the ***ALU***. Named mapping allows you to have freedom with the order of port declarations.
      alu alu(.clk(clk), .opcode(opcode), …….);
   ii. Data type declaration, "**reg**" or "**wire**" or others.
   iii. Modeling the clock generator. This is a very efficient way to model a structural clock. If your design is modeled structurally. It will simulate faster with a structural clock.
      a. the clock has a period of *20ns* : 10ns high and 10ns low.
      b. the initial clock resets the clock to a unknown state.



time	clk
0	x
10	1
20	0

iv. Setup the waveform display.

```
initial  begin
         $shm_open("alu.shm");   // SHM Database
         $shm_probe("AC");
         $fsdbDumpfile("alu.fsdb");   // FSDB Database
         $fsdbDumpvars();
          end
```

v. Apply stimulus as follows :

a. Assert and deassert *reset* to test that the ALU reset when reset = 1.

b. You can verify operation for all 8 opcodes, you can use "for loop" to generate opcodes automaticly.

c. You can verify operation with unknown opcode

d. You can verify operation of zero bit

Note: You can provide stimulus from an initial block in the testbench.

vi. Display the result of ALU's outputs when stimulus input to the ALU.

3. using the *alu_test.v* file to test the ALU file *alu.v*
simulate with *verilog-XL* , enter :

```
verilog  alu_test.v  alu.v
```

If you using *NC-Verilog*, enter :

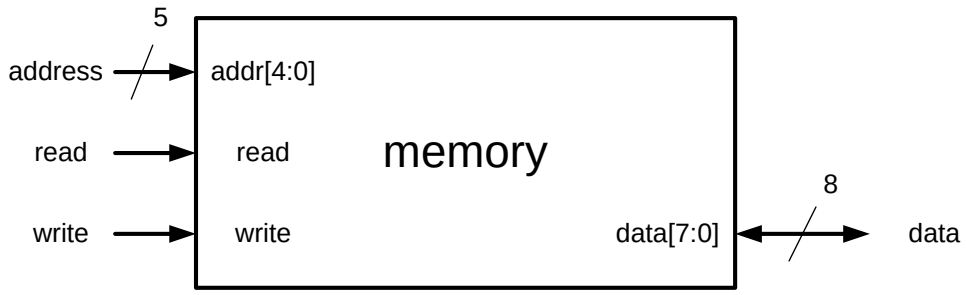
```
ncverilog  alu_test.v  alu.v  +access+r
```

4. Check the result.

End of Lab5

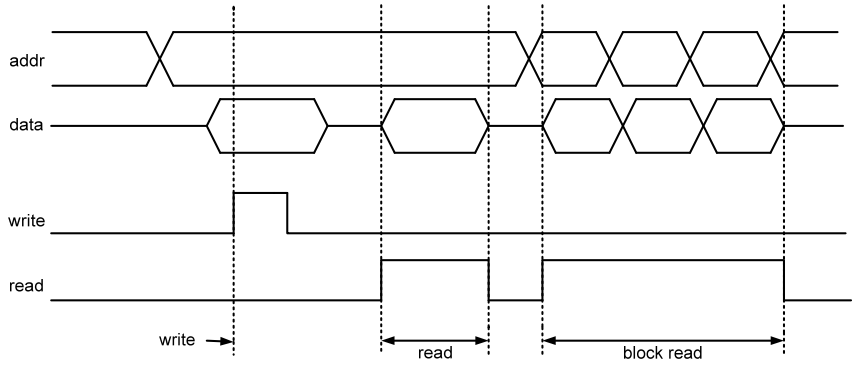
Lab6 : Memory

- Modeling a Memory with a Bidirectional Data Bus.
- Specifications
  - Module name : mem
  - Input pins : addr[4:0], read, write
  - Output pins : data[7:0]
  - Function :



1. Change to the ***Lab6*** directory. This directory contains two file:
   i. The *mem_test.v* file is the testbench for the *memory* model. It has errors that you will correct during this lab.
   ii. The *mem.v* file exists. But only the module header has been entered for you. The memory data bus is bidirectional.

2. Edit the file *mem.v* to finish modeling the memory design.
   i. The name of the memory register array is *memory*.
   The MSB of each word is bit *7*.
   The LSB of each word is bit *0*.
   The first address is address *0*.
   The last address is address *31* (hex 1F).
   ii. Access to the memory model is *asynchronous*, and is controlled by two control lines : *write* and *read*.



iii. At the positive edge of the *write* control line, the value on the *data* bus is written into the memory location addressed by the *addr* bus.

Note : You cannot make a *procedural assignment* to a net data type, such as wire or tri. However, signal connected to inout ports must be of a net data type.

iv. When the *read* line is asserted (logic 1), the contents of the memory location addressed by the *addr* bus are continually driven to the *data* bus.

Note : You can use either a *continuous assignment* or *bufif primitives*. But *continuous assignments* are easier understand.

v. The memory is capable of block reads : if the *addr* bus changes while *read* is asserted, the contents of the new location addressed are immediately transferred to the *data* bus.

vi. When the *read* control line is not asserted (logic 0), the memory places the *data* bus in a high-impedance state.

3. Test your memory design using the *mem_test.v* and observing the errors information that reported.
   i. Why is the bidirectional data bus declared as data type *wire*?
   ii. What would happed if data were declared as data type *reg*, so that the stimulus could use *procedural assignments* to place values on the data bus?

4. Correct these errors by modifying the *mem_test.v* file. Then run the simulation again to verify the correct functionality of the memory.

HINT : use the *shadow register* for the *procedural assignment*:

a. Declare a register for *procedural assignment*, or writes, to the data bus.

b. Use a *continuous assignment* with the conditional operator to transfer the register to the data bus when *read* is deasserted. Otherwise, Tri-state the data bus.

5. Check the result.

```
Setting all memory cells to zero…
Reading from one memory address…
Setting all memory cells to alternating patterns…
Doing block read from five memory addresses…

Completed Memory Tests With 0 Errors!
```

End of Lab6

Lab7 : FSM

- Please design a serial input bit-stream pattern detector module.
- Specifications
  - Module name : fsm_bspd
  - Input pins : clk, reset, bit_in
  - Output pins : det_out
  - Function : serial input bit-stream pattern detector.
    Using finite state Mealy-machine. "*det_out*" is to be low(logic 0), unless the input bit-stream is "0010" sequentially. Example :
    bit_in : ..... 0 0 1 0 0 1 0 0 0 1 0...
    det_out : .... 0 0 0 1 0 0 1 0 0 0 1 0...

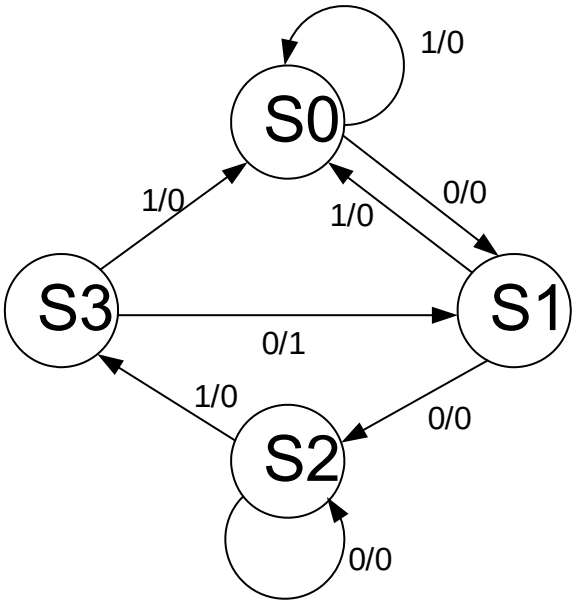
1. Change to the ***Lab7*** directory. This directory contains two file:
   i. The *fsm_test.v* file is the testbench for the ***fsm*** module.
   ii. The *fsm_bspd.v* file exists. But only the module header has been entered for you.

2. Edit the file *fsm_bspd.v* to finish modeling the serial bit stream pattern detector using **finite state Mealy-machine**.
   i. All inputs are synchronized at clock(clk) rising edge.
   ii. It is synchronous reset architecture.
   iii. It can use any one of following design styles to implement the finite state machine
      a. Separate CS, NS and OL
      b. Combine CS and NS, separate OL
      c. Combine NS and OL, separate CS
   iv. The state assignment is as below:
      a. less than 4 states : S0->00, S1->01, S2->10, S3->11
      b. between 5 and 8 states : S0->000, S1->001, S2->010, S3->011, S4->100, S5->101, S6->110, S7->111

3. Test your fsm design using the *fsm_test.v*.
   verilog fsm_test.v fsm_bspd.v
   or ncverilog fsm_test.v fsm_bspd.v +access+r

4. Maybe you can use these state diagram and state table to finish your finite state Mealy-Machine.

State Diagram



State Table

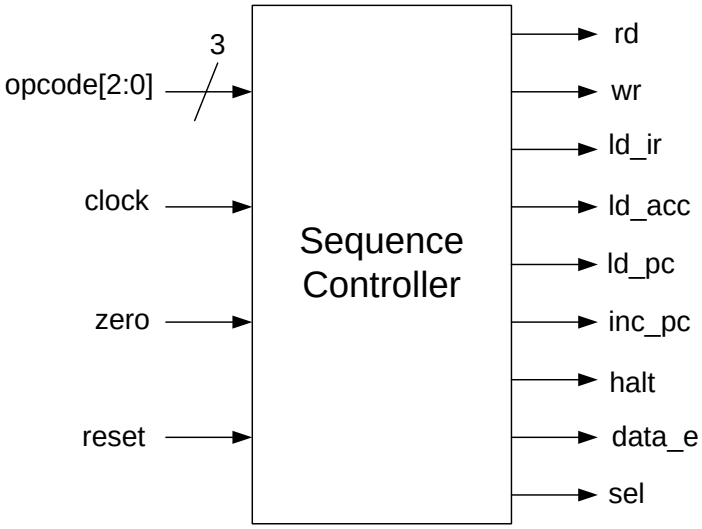
Present state	Next state		Present output	
	X = 0	X = 1	X = 0	X = 1
S0	S1	S0	0	0
S1	S2	S0	0	0
S2	S2	S3	0	0
S3	S1	S0	1	0

5. Check the result.

End of Lab7

Lab8 : Sequence Controller

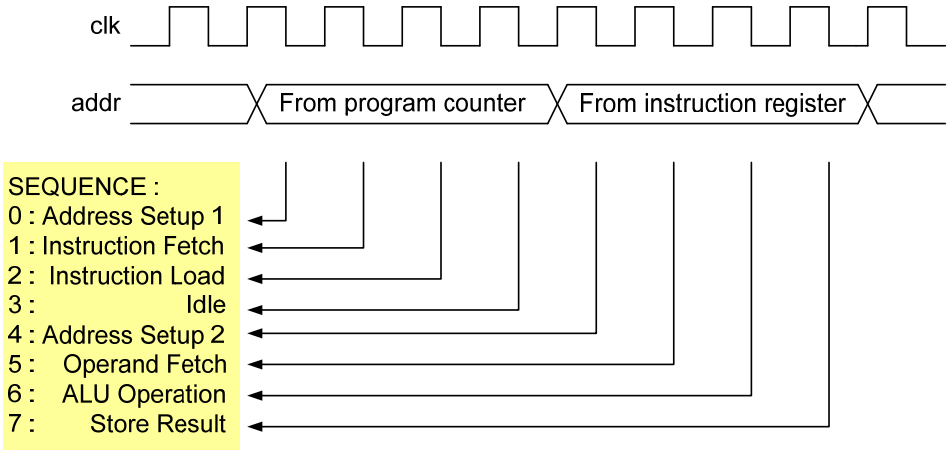
- Model a Sequence Controller and test it, using test vectors and expected response files.
- Specifications
  - Module name : control
  - Input pins : clk, rst, zero, opcode[2:0]
  - Output pins : rd, wr, ld_ir, ld_acc, ld_pc, inc_pc, halt, data_e, sel .
  - Function :



1. Change to the ***Lab8*** directory. This directory contains four file:
   1). The *control_test.v* file is the testbench for checking the controller design.
   2). The *control.v* file exists. The module and port declarations of the sequence controller.
   3). *test_vectors.pat* is the file with input test vectors
   4). *expected_results.pat* is the pattern file of the expected outputs.

2. Read the specifications that follow, making sure you understand the controller design. Following the specifications are brief instructions for modeling the controller at the RTL level.
   1). The controller is synchronized at clock(*clk*) rising edge.
   2). It is asynchronous reset architecture, low-assertion reset.

3). The controller process a sequence of eight steps to fetch and execute each instruction. This sequence of eight steps is called a fetch cycle.



The instruction fetched from memory during the 1st half of a fetch cycle determines what operation to perform during the 2nd half of the fetch cycle.

4). The eight possible values of the *opcode* make up the instruction set for the VeriRisc CPU:

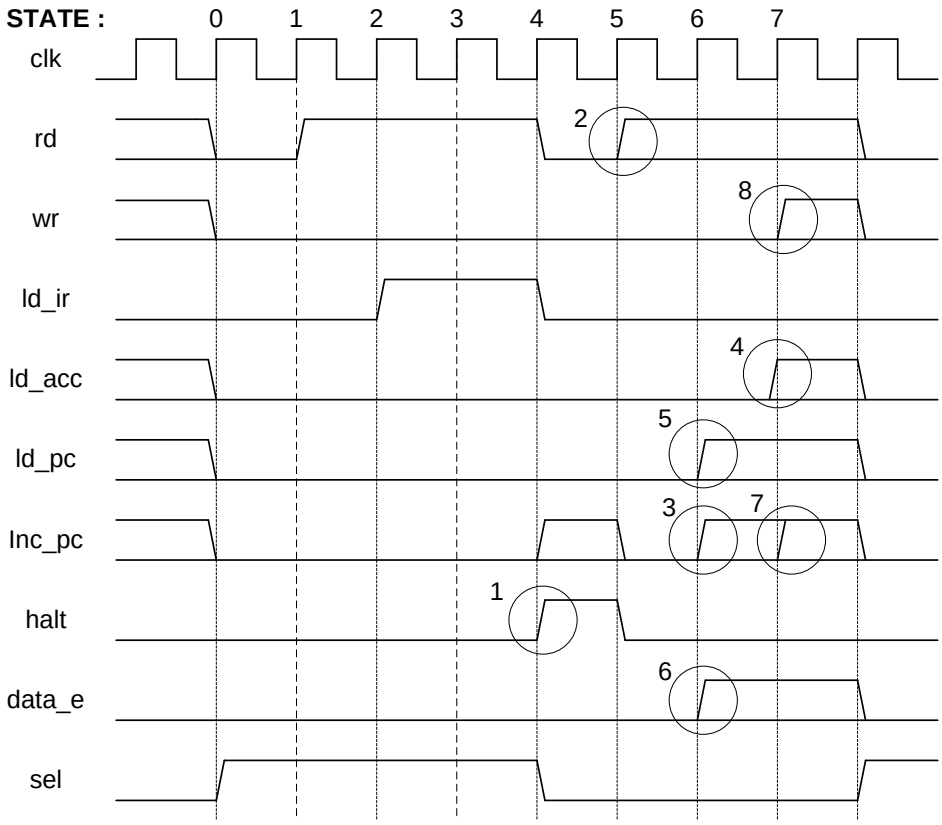
The VeriRisc CPU instruction Set

INSTRUCTION	OPCODE
HLT (halt)	000
SKZ (skip if zero bit set)	001
ADD (data + accumulator)	010
AND (data & accumulator)	011
XOR (data ^ accumulator)	100
LDA (load accumulator)	101
STO (store accumulator)	110
JMP (jump to new address)	111

In each of the eight steps, the controller asserts and deasserts nine 1-bit control signals, based upon the value of the instruction *opcode* and the *zero* bit from an ALU, These control signals are the outputs from the controller.

5). The *opcode* and *zero* inputs are always stable before any clock edge.

6). The following waveform diagram shows the behavior of the controller. Each circled signal transition occurs under conditions explained below.



Note :

a. *halt* will be asserted if the instruction is HLT.
b. *rd* will be asserted for ADD, AND, XOR, and LDA instruction.
c. *inc_pc* will be asserted if instruction is SKZ and the zero bit is set.
d. *ld_acc* will be asserted for ADD, AND, XOR and LDA instructions.
e. *ld_pc* will be asserted if instruction is JMP.
f. *data_e* wil be asserted unless instruction is ADD, AND, XOR or LDA.
g. *inc_pc* will be asserted if instruction is JMP.
h. *wr* will be asserted if instruction is STO.

3. Because timing diagrams are often difficult to understand, you can instead refer to the following state table for the behavior of the controller:

rst	state	rd	wr	ld_ir	ld_acc	ld_pc	inc_pc	halt	data_e	sel
0	**? : Reset**	0	0	0	0	0	0	0	0	0
1	**0 : Address Setup 1**	0	0	0	0	0	0	0	0	1
1	**1 : Instruction Fetch**	1	0	0	0	0	0	0	0	1
1	**2 : Instruction Load**	1	0	1	0	0	0	0	0	1
1	**3 : Idle**	1	0	1	0	0	0	0	0	1
1	**4 : Address Setup 2**									
	if HLT opcode	0	0	0	0	0	1	1	0	0
	all other opcodes	0	0	0	0	0	1	0	0	0
1	**5 : Operand Fetch**									
	ADD,AND,XOR,LDA	1	0	0	0	0	0	0	0	0
	all other opcodes	0	0	0	0	0	0	0	0	0
1	**6 : ALU Operation**									
	SKZ and zero set	0	0	0	0	0	1	0	1	0
	ADD,AND,XOR,LDA	1	0	0	0	0	0	0	0	0
	JMP	0	0	0	0	1	0	0	1	0
	all other opcodes	0	0	0	0	0	0	0	1	0
1	**7 : Store Result**									
	SKZ and zero set	0	0	0	0	0	1	0	1	0
	ADD,AND,XOR,LDA	1	0	0	1	0	0	0	0	0
	STO	0	1	0	0	0	0	0	1	0
	JMP	0	0	0	0	1	1	0	1	0
	all other opcodes	0	0	0	0	0	0	0	1	0

4. Modeling the Sequence Controller. The behavior of a sequencer such as this one is best characterized as a state machine. More specifically, because this example does not contain branched, it is referred to as a state counter.
   Model the state machine explicitly. Although in Verilog you can model a simple state machine(like this one) either implicitly or explicitly, implicit state machine are rarely accepted by synthesis tools. They are generally more difficult to code, understand, and modify.

5. Simulation the Design. To run a simulation to test your controller model and correct any errors, enter :

   verilog control_test.v control.v

   or nverilog control_test.v control.v +access+r

6. When you reach the end of the text vectors. If *0* errors have been detected, you have successfully modeled your sequence controller.

End of Lab8