Advanced Computer Architecture

Lecture No. 6

Reading Material

Handouts Slides

Summary

- Using Behavioral RTL to Describe the SRC (continued)
- Implementing Register Transfer using Digital Logic Circuits

Using behavioral RTL to Describe the SRC (continued)

Once the instruction is fetched and the PC is incremented, execution of the instruction starts. In the following discussion, we denote instruction fetch by "iF" and instruction execution by "iE".

```
iE:= (
        (op<4..0>= 1) : R [ra] ←  M [disp],
        (op<4..0>= 2) : R [ra] ←  M [rel],
                        . . .
                        . . .
        (op<4..0>=31) : Run ←  0,); iF);
```

As shown above, instruction execution can be described by using a long list of conditional operations, which are inherently "disjoint". Only one of these statements is executed, depending on the condition met, and then the instruction fetch statement (iF) is invoked again at the end of the list of concurrent statements. Thus, instruction fetch (iF) and instruction execution statements invoke each other in a loop. This is the fetch-execute cycle of the SRC.

Concurrent Statements

The long list of concurrent, disjoint instructions of the instruction execution (iE) is basically the complete instruction set of the processor. A brief overview of these instructions is given below:

Load-Store Instructions

(op<4..0>= 1) : R [ra] ← M [disp], load register (ld)

This instruction is to load a register using a displacement address specified by the instruction, i.e., the contents of the memory at the address 'disp' are placed in the register R [ra].

(op<4..0>= 2) : R [ra] ← M [rel], load register relative (ldr)

If the operation field 'op' of the instruction decoded is 2, the instruction that is executed is loading a register (target address of this register is specified by the field ra) with memory contents at a relative address, 'rel'. The relative address calculation has been explained in this section earlier.

(op<4..0>= 3) : M [disp] ← R [ra], store register (st)

If the op-code is 3, the contents of the register specified by address ra, are stored back to the memory, at a displacement location 'disp'.

(op<4..0>= 4) : M[rel] ← R[ra], store register relative (str)

If the op-code is 4, the contents of the register specified by the target register address ra, are stored back to the memory, at a relative address location 'rel'.

(op<4..0>= 5) : R [ra] ← disp, load displacement address (la)

For op-code 5, the displacement address disp is loaded to the register R (specified by the target register address ra).

(op<4..0>= 6) : R [ra] ← rel, load relative address (lar)

For op-code 6, the relative address rel is loaded to the register R (specified by the target register address ra).

Branch Instructions

(op<4..0>= 8) : (cond : PC ← R [rb]), conditional branch (br)

If the op-code is 8, a conditional branch is taken, that is, the program counter is set to the target instruction address specified by rb, if the condition 'cond' is true.

(op<4..0>= 9) : (R [ra] ← PC,
cond : (PC ← R [rb])), branch and link (brl)

If the op field is 9, branch and link instruction is executed, i.e. the contents of the program counter are stored in a register specified by ra field, (so control can be returned to it later), and then the conditional branch is taken to a branch target address specified by rb. The branch and link instruction is useful for returning control to the calling program after a procedure call returns.

The conditions that these 'conditional' branches depend on, are specified by the field c3 that has 3 bits. This simply means that when c3<2..0> is equal to one of these six values, we substitute the expression on the right hand side of the : in place of cond.

These conditions are explained here briefly.

cond := (

c3<2..0>=0 : 0, never

If the c3 field is 0, the branch is never taken.

c3<2..0>=1 : 1, always

If the field is 1, branch is taken

c3<2..0>=2 : R [rc]=0, if register is zero

If c3 = 2, a branch is taken if the register rc = 0.

c3<2..0>=3 : R [rc] ≠ 0, if register is nonzero

If c3 = 3, a branch is taken if the register rc is not equal to 0.

c3<2..0>=4 : R [rc]<31>=0 if positive or zero

If c3 is 4, a branch is taken if the register value in the register specified by rc is greater than or equal to 0.

c3<2..0>=5 : R [rc]<31>=1), if negative

If c3 = 5, a branch is taken if the value stored in the register specified by rc is negative.

Arithmetic and Logical instructions

(op<4..0>=12) : R [ra] ← R [rb] + R [rc],

If the op-code is 12, the contents of the registers rb and rc are added and the result is stored in the register ra.

(op<4..0>=13) : R [ra] ← R [rb] + c2<16..0> {sign extended},

If the op-code is 13, the content of the register rb is added with the immediate data in the field c2, and the result is stored in the register ra.

(op<4..0>=14) : R [ra] ← R [rb] – R [rc],

If the op-code is 14, the content of the register rc is subtracted from that of rb, and the result is stored in ra.

(op<4..0>=15) : R [ra] ← -R [rc],

If the op-code is 15, the content of the register rc is negated, and the result is stored in ra.

(op<4..0>=20) : R [ra] ← R [rb] & R [rc],

If the op field equals 20, logical AND of the contents of the registers rb and rc is obtained and the result is stored in register ra.

(op<4..0>=21) : R [ra] ← R [rb] & c2<16..0> {sign extended},

If the op field equals 21, logical AND of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.

(op<4..0>=22) : R [ra] ← R [rb] ~ R [rc],

If the op field equals 22, logical OR of the contents of the registers rb and rc is obtained and the result is stored in register ra.

(op<4..0>=23) : R [ra] ← R [rb] ~ c2<16..0> {sign extended},

If the op field equals 23, logical OR of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.

(op<4..0>=24) : R [ra] ← !R [rc],

If the op-code equals 24, the content of the logical NOT of the register rc is obtained, and the result is stored in ra.

Shift instructions

(op<4..0>=26): R [ra]<31..0 > ← (n α 0) © R [rb] <31..n>,

If the op-code is 26, the contents of the register rb are shifted right n bits times. The bits that are shifted out of the register are discarded. 0s are added in their place, i.e. n number of 0s is added (or concatenated) with the register contents. The result is copied to the register ra.

(op<4..0>=27) : R [ra]<31..0 > ← (n α R [rb] <31>) © R [rb] <31..n>,

For op-code 27, shift arithmetic operation is carried out. In this operation, the contents of the register rb are shifted right n times, with the most significant bit, i.e., bit 31, of the register rb added in their place. The result is copied to the register ra.

(op<4..0>=28) : R [ra]<31..0 > ← R [rb] <31-n..0> © (n α 0),

For op-code 28, the contents of the register rb are shifted left n bits times, similar to the shift right instruction. The result is copied to the register ra.

(op<4..0>=29) : R [ra]<31..0 > ← R [rb] <31-n..0> © R [rb]<31..32-n >,

The instruction corresponding to op-code 29 is the shift circular instruction. The contents of the register rb are shifted left n times, however, the bits that move out of the register in the shift process are not discarded; instead, these are shifted in from the other end (a circular shifting). The result is stored in register ra.

where

n := ((c3<4..0>=0) : R [rc],

(c3<4..0>!=0) : c3 <4..0>),

Notation:

α means replication

© means concatenation

Miscellaneous instructions

(op<4..0>= 0) , No operation (nop)

If the op-code is 0, no operation is carried out for that clock period. This instruction is used as a stall in pipelining.

(op<4..0>= 31) : Run ← 0, Halt the processor (Stop)

); iF);

If the op-code is 31, run is set to 0, that is, the processor stops execution.

After one of these disjoint instructions is executed, iF, i.e. instruction Fetch is carried out once again, and so the fetch-execute cycle continues.

Implementing Register Transfers using Digital Logic Circuits

We have studied the register transfers in the previous sections, and how they help in implementing assembly language. In this section we will review how the basic digital logic circuits are used to implement instructions register transfers. The topics we will cover in this section include:

1. A brief (and necessary) review of logic circuits
2. Implementing simple register transfers
3. Register file implementation using a bus
4. Implementing register transfers with mathematical operations
5. The Barrel Shifter
6. Implementing shift operations

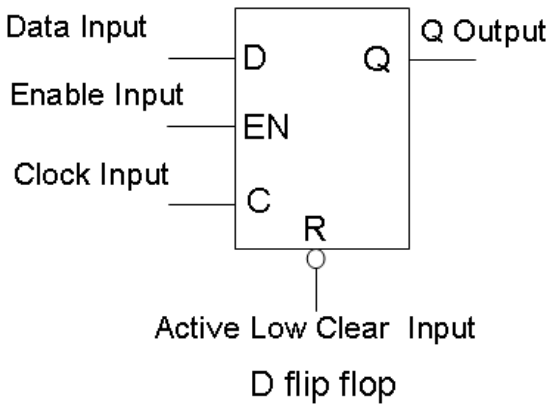
Review of logic circuits

Before we study the implementation of register transfers using logic circuits, a brief overview of some of the important logic circuits will prove helpful. The topics we review in this section include

1. The basic D flip flop
2. The n-bit register
3. The n-to-1 multiplexer
4. Tri-state buffers

The basic D flip flop

A flip-flop is a bi-stable device, capable of storing one bit of Information. Therefore, flip-flops are used as the building blocks of a computer's memory as well as CPU registers.



D flip flop

There are various types of flip-flops; most common type, the D flip-flop is shown in the figure given. The given truth table for this positive-edge triggered D flip-flop shows that the flip-flop is set (i.e. stores a 1) when the data input is high on the leading (also called the positive) edge of the clock; it is reset (i.e., the flip-flop stores a 0) when the data input is 0 on the leading edge of the clock. The clear input will reset the flip-flop on a low input.

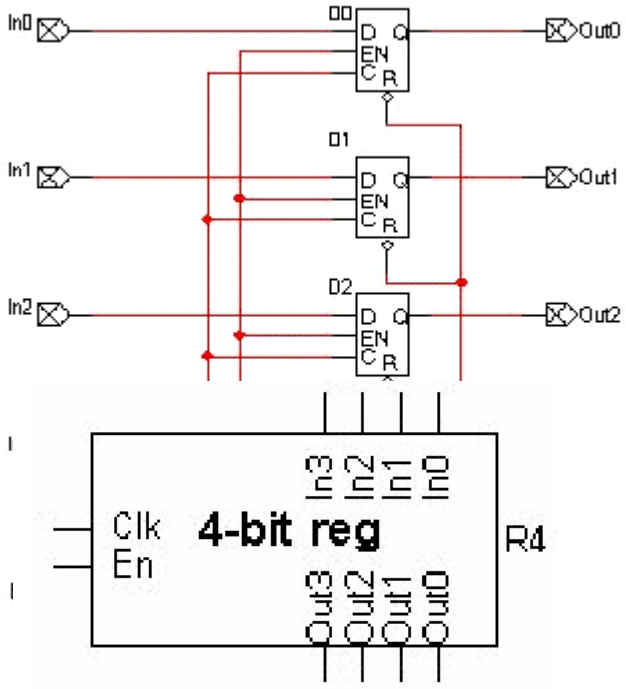
The n-bit register

A n-bit register can be formed by grouping n flip-flops together. So a register is a device in which a group of flip-flops operate synchronously.

A register is useful for storing binary data, as each flip-flop can store one bit. The clock input of the flip-flops is grouped together, as is the enable input. As shown in the figure, using the input lines a binary number can be stored in the register by applying the corresponding logic level to each of the flip-flops simultaneously at the positive edge of the clock.

The next figure shows the symbol of a 4-bit register used for an integrated circuit. In0 through In3 are the four input lines, Out0 through Out3 are the four output lines, Clk is the clock input, and En is the enable line. To get a better understanding of this register, consider the situation where we want to store the binary number 1000 in the register. We will apply the number to the input lines, as shown in the figure given.

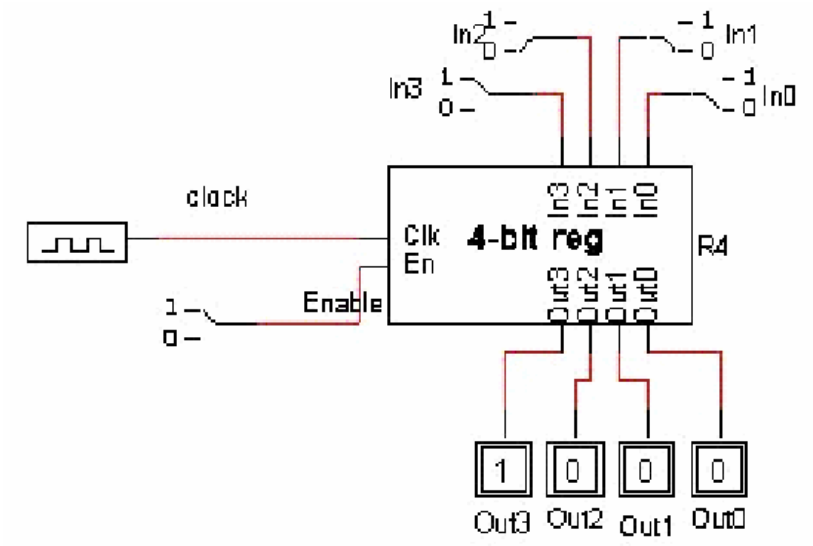
EN	D	Q
0	X	X



4-bit Register Symbol

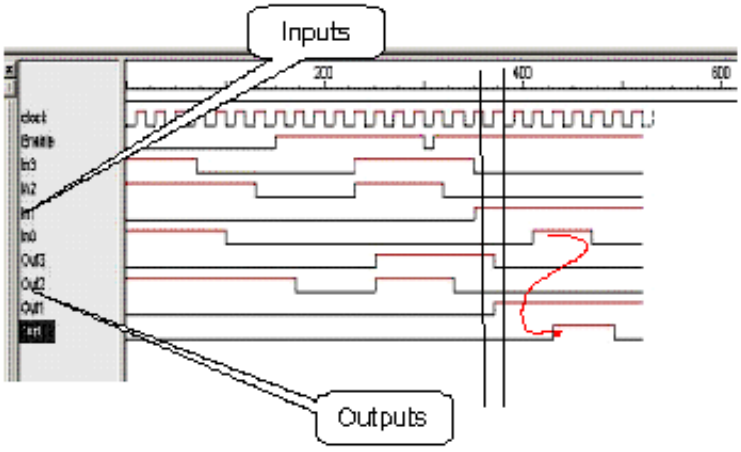
On the leading edge of the clock, the number will be stored in the register. The enable input has to be high if the number is to be stored into the register.

.



test circuit for 4-bit register

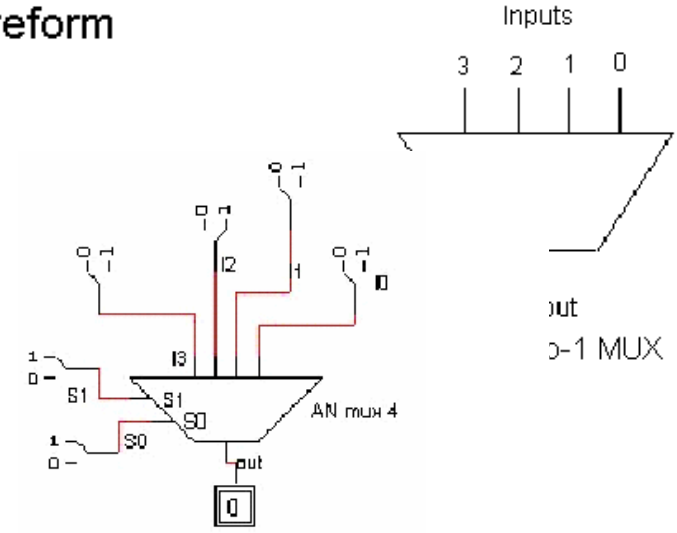
Waveform/Timing diagram



Timing waveform

The n-to-1 multiplexer

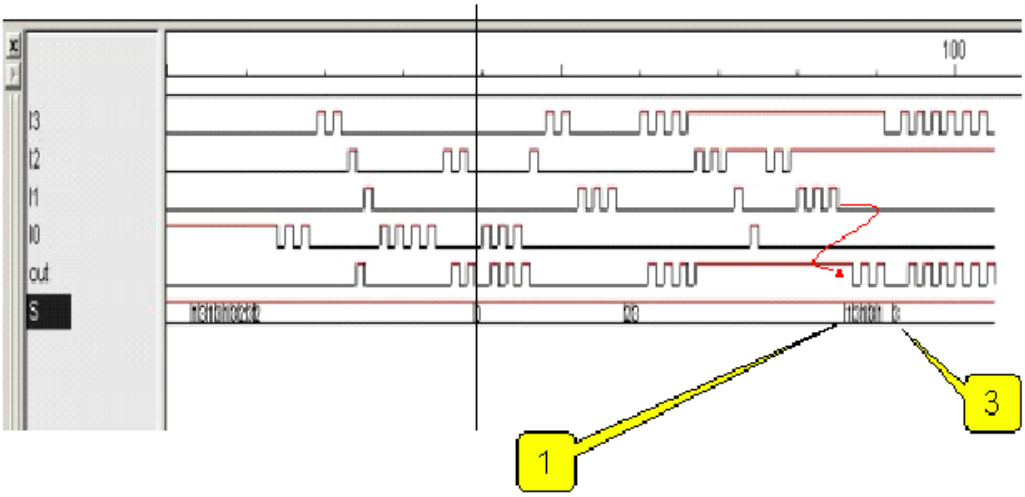
A multiplexer is a device, constructed through combinational logic, which takes n inputs and transfers one of them as the output at a time. The input that is selected as the output depends on the selection lines, also called the control input lines. For an n-to-1



test circuit for 4-to-1 MUX

multiplexer, there are n input lines, $\log_2 n$ control lines, and 1 output line. The given figure shows a 4-to-1 multiplexer. There are 4 input lines; we number these lines as line 0 through line 3. Subsequently, there are 2 select lines (as $\log_2 4 = 2$).
For a better understanding, let us consider a case where we want to transfer the input of line 3 to the output of the multiplexer. We will need to apply the binary number 11 on the select lines (as the binary number 11 represents the decimal number 3). By doing so, the output of the multiplexer will be the input on line 3, as shown in the test circuit given.

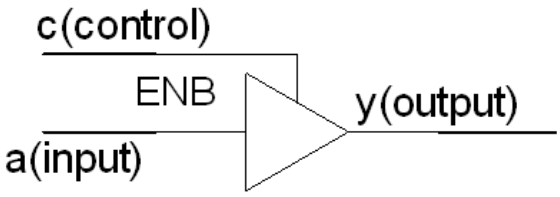
Timing waveform



Timing Waveform for MUX

Tri-state buffers

The tri-state buffer, also called the three-state buffer, is another important component in the digital logic domain. It has a single input, a single output, and an enable line. The input is concatenated to the output only if it is enabled through the enable line, otherwise it gives a high impedance output, i.e. it is tri-stated, or electrically disconnected from the input These buffers are available both in the inverting and the non-inverting form. The inverting tri-state buffers output the 'inverted' input when they are enabled, as opposed to their non-inverting counterparts that simply output the input when enabled. The circuit symbol of the tri-state buffers is shown. The truth table



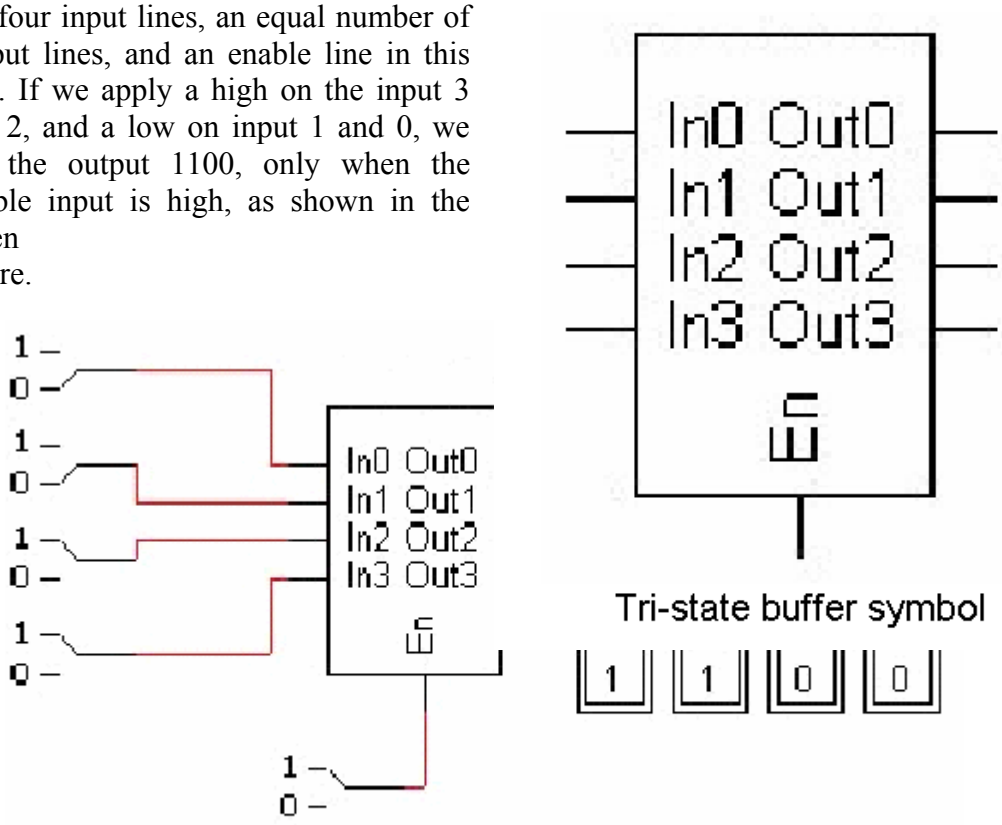
Tri-state buffer

c	a	y
0	0	Z
0	1	Z
1	0	0
1	1	1

Truth table :Tri-state buffer

further clarifies the working of a non-inverting tri-state buffer.

We can see that when the enable input (or the control input) c is low (0), the output is high impedance Z. The symbol of a 4-bit tri-state buffer unit is shown in the figure. There are four input lines, an equal number of output lines, and an enable line in this unit. If we apply a high on the input 3 and 2, and a low on input 1 and 0, we get the output 1100, only when the enable input is high, as shown in the given figure.



Test circuit for Tri-state buffer

Implementing simple register transfers

We now build on our knowledge of the primitive logic circuits to understand how register transfers are implemented. In this section we will study the implementation of the following

- Simple conditional transfer
- Concept of control signals
- Two-way transfers
- Connecting multiple registers
- Buses
- Bus implementations

Simple conditional transfer

In a simple conditional transfer, a condition is checked, and if it is true, the register transfer takes place. Formally, a conditional transfer is represented as

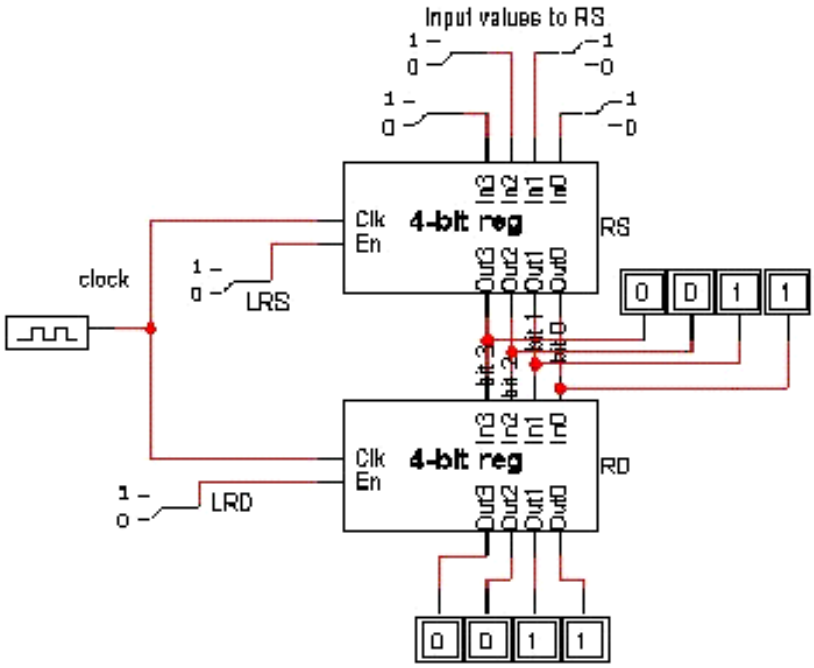
Cond: RD ← RS

This means that if the condition 'Cond' is true, the contents of the register named RS (the source register) are copied to the register RD (the destination register). The following figure shows how the registers may be interconnected to achieve a conditional transfer. In

this circuit, the output of the source register RS is connected to the input of the destination registers RD. However, notice that the transfer will not take place unless the enable input of the destination register is activated. We may say that the 'transfer' is being controlled by the enable line (or the control signal). Now, we are able to control the transfer by selectively enabling the control signal, through the use of other combinational logic that may be the equivalent of our condition. The condition is, in general, a Boolean expression, and in this example, the condition is equivalent to LRD =1.

Two-way transfers

In the above example, only one-way transfer was possible, i.e., we could only copy the contents of RS to RD if the condition was met. In order to be able to achieve two-way transfers, we must also provide a path from the output of the register RD to input of register RS. This will enable us to implement



Conditional Trasnfer

Cond1: RD ← RS

Cond2: RS ← RD

Connecting multiple registers

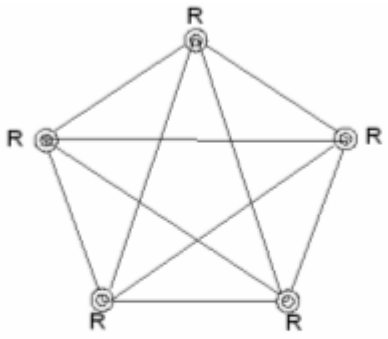
We have seen how two registers can be connected. However, in a computer we need to connect more than just two registers. In order to connect these registers, one may argue that a connection between the input and output of each be provided. This solution is shown for a scenario where there are 5 registers that need to be interconnected.

We can see that in this solution, an m-bit register requires two connections of m-wires each. Hence five m-bit registers in a "point-to-point" scheme require 20 connections; each with m wires. In general, n registers in a point to point scheme require n (n-1) connections. It is quite obvious that this solution is not going to scale well for a large

number of registers, as is the case in real machines. The solution to this problem is the use of a bus architecture, which is explained in the following sections.

Buses

A bus is a device that provides a shared data path to a number of devices that are connected to it, via a 'set of wires' or a 'set of conductors'. The modern computer systems extensively employ the bus architecture. Control signals are needed to decide which two entities communicate using the shared medium, i.e. the bus, at any given time. This control signals can be open collector gate based, tri-state buffer based, or they can be implemented using multiplexers.

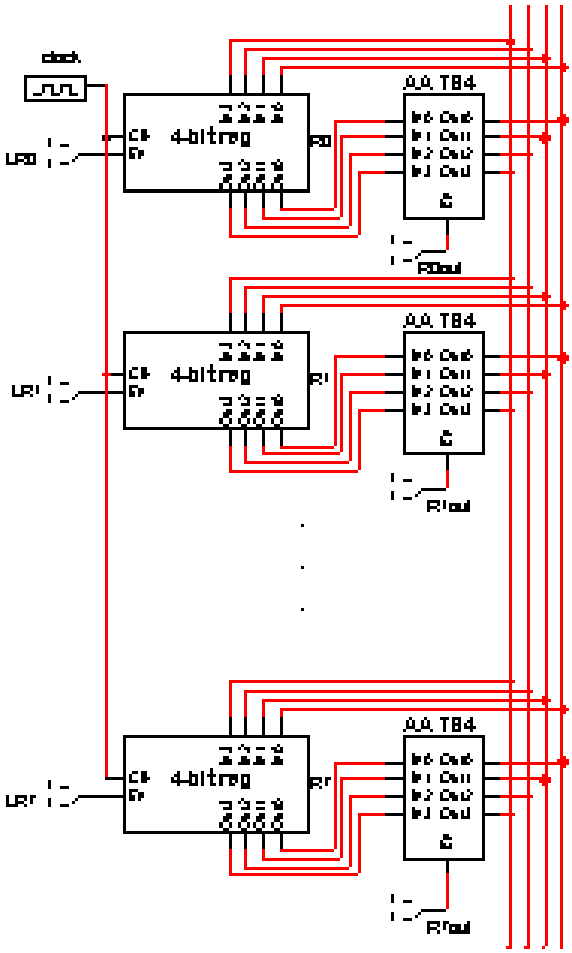


Multiple register connections

Register file implementation using the bus architecture

A number of registers can be inter-connected to form a register file, through the use of a bus. The given diagram shows eight 4-bit registers (R0, R1, …, R7) interconnected through a 4-bit bus using 4-bit tri-state buffer units (labeled AA_TS4). The contents of a particular register can be transferred onto the bus by applying a logical high input on the enable of the corresponding tri-state buffer. For instance, R1out can be used to enable the tri-state buffers of the register R1, and in turn transfer the contents of the register on the bus.

Once the contents of a particular register are on the bus, the contents may be transferred, or read into any other register. More than one register may be written in this manner; however, only one register can write its value on the bus at a given time.



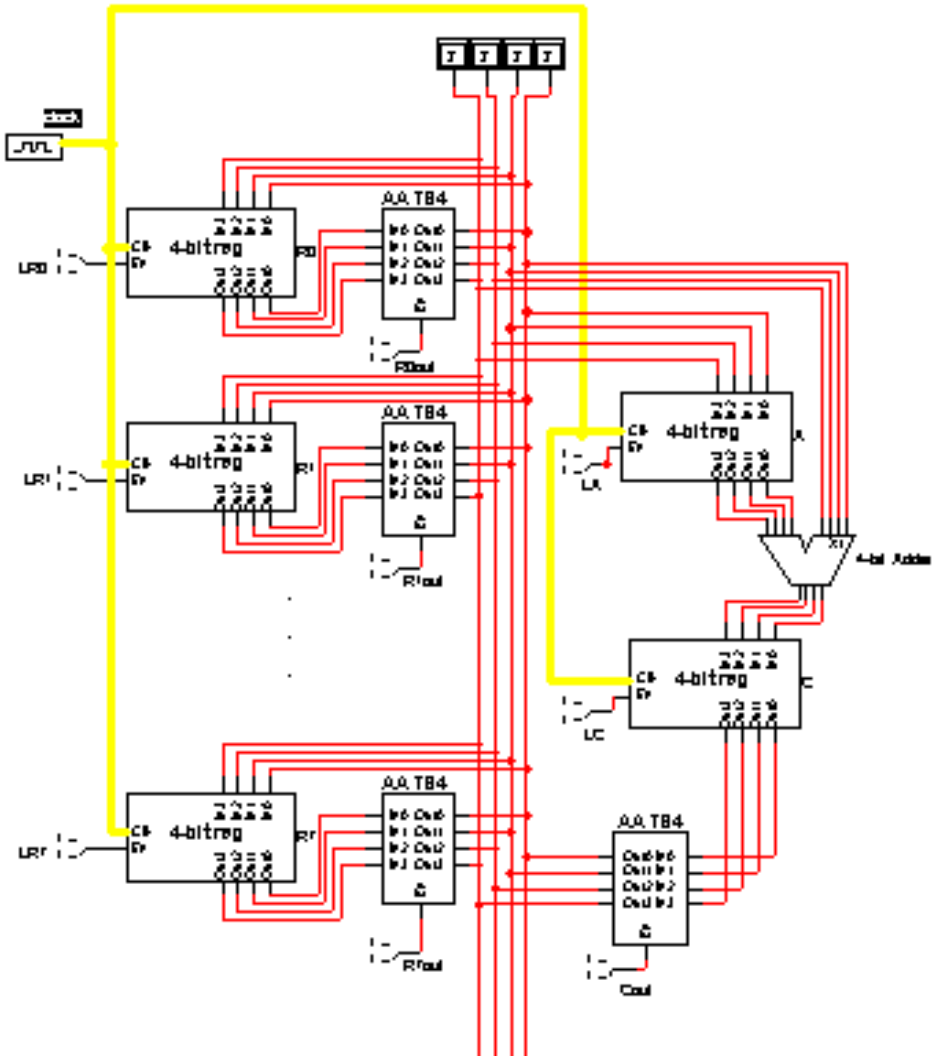
Register File

Implementing register transfers with mathematical operations

We have studied the implementation of simple register transfers; however, we frequently encounter register transfers with mathematical operations. An example is

(opc=1): R4← R3 + R2;

These mathematical operations may be achieved by introducing appropriate combinational logic; the above operation can be implemented in hardware by including a 4-bit adder with the register files connected through the bus. There are two more registers in this configuration, one for holding one of the operands, and the other for holding the result before it is transferred to the destination register. This is shown in the figure below.



We now take a look at the steps taken for the (conditional, mathematical) transfer (opc=1): R4← R3 + R2. First of all, if the condition opc = 1 is met, the contents of the first operand register, R3, are transferred to the temporary register A through the bus. This is done by activating R3out. It lets the contents of the register R3 to be loaded on the bus. At the same time, applying a logical high input to LA enables the load for the register A. This lets the binary number on the bus (the contents of register R3) to be loaded into the register A. The next step is to enable R2out to load the contents of the register R2 onto the bus. As can be observed from the figure, the output of the register A is one of the inputs to the 4-bit adder; the other input to the adder is the bus itself. Therefore, as the contents of register R2 are loaded onto the bus, both the operands are available to the adder. The output can then be stored to the register RC by enabling its write. So a high input is applied to LC to store the result in register RC.

Time step	Operation to be performed (structural RTL)	Control signals to be activated
1	A ← R3	LA, R3out
2	C ← A + R2	LC, R2out
3	R4 ← C	LR4, Cout

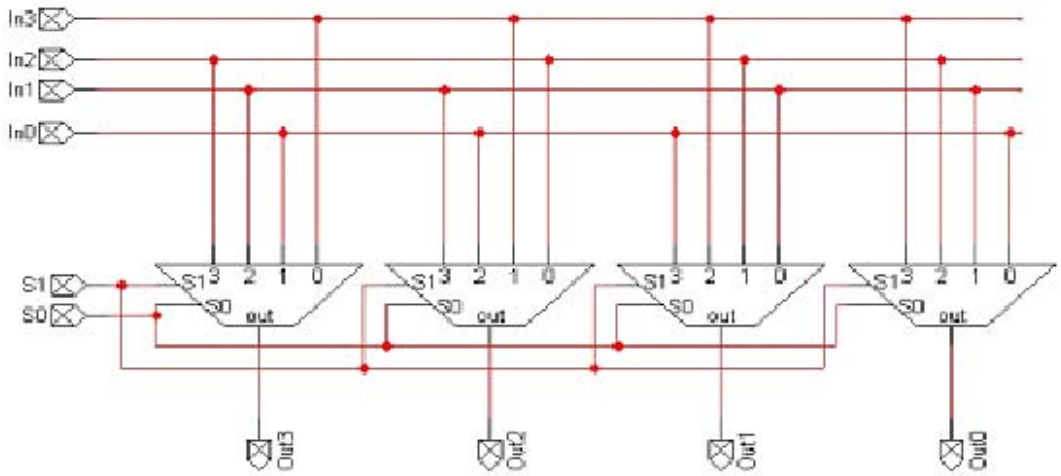
Structural RTL: add operation

The third and final step is to store (transfer) the resultant number in the destination register R4. This is done by enabling Cout, which writes the number onto the bus, and then enabling the read of the register R4 by activating the control signal to LR4. These steps are summarized in the given table.

The barrel shifter

Shift operations are frequently used operations, as shifts can be used for the implementation of multiplication and division etc. A bi-directional shift register with a parallel load capability can be used to perform shift operations. However, the delays in such structures are dependent on the number of shifts that are to be performed, e.g., a 9 bit shift requires nine clock periods, as one shift is performed per clock cycle. This is not an optimal solution. The barrel shifter is an alternative, with any number of shifts accomplished during a single clock period. Barrel shifters are constructed by using multiplexers. An n-bit barrel shifter is a combinational circuit implemented using n multiplexers. The barrel provides a shifted copy of the input data at its output. Control inputs are provided to specify the number of times the input data is to be shifted. The shift process can be a simple one with 0s used as fillers, or it can be a rotation of the input data. The corresponding figure shows a barrel shifter that shifts right the input data; the number of shifts depends on the bit pattern applied on the control inputs S0, S1.

The function table for the barrel shifter is given. We see from the table that in order to apply single shift to the input number, the control signal is 01 on (S1, S0), which is the binary equivalent of the decimal number 1. Similarly, to apply 2 shifts, control signal 10



Barrel Shifter

(on S1, S0) is applied; 10 is the binary equivalent of the decimal number 2. A control input of 11 shifts the number 3 places to the right.

Now we take a look at an example of the shift operation being implemented through the use of the barrel shifter:

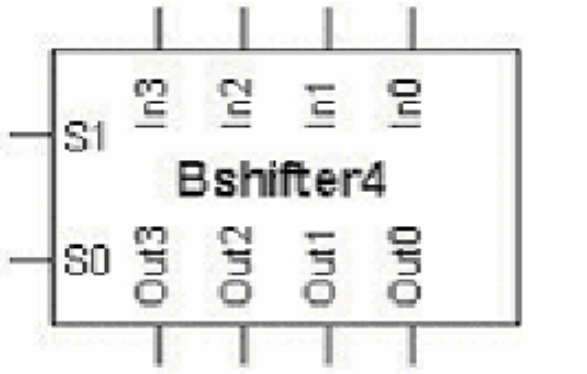
R4← ror R3 (2 times);

The shift functionality can be incorporated into the register file circuit with the bus architecture we have been building, by introducing the barrel shifter, as shown in the given figure.

To perform the operation,

R4← ror R3 (2 times),

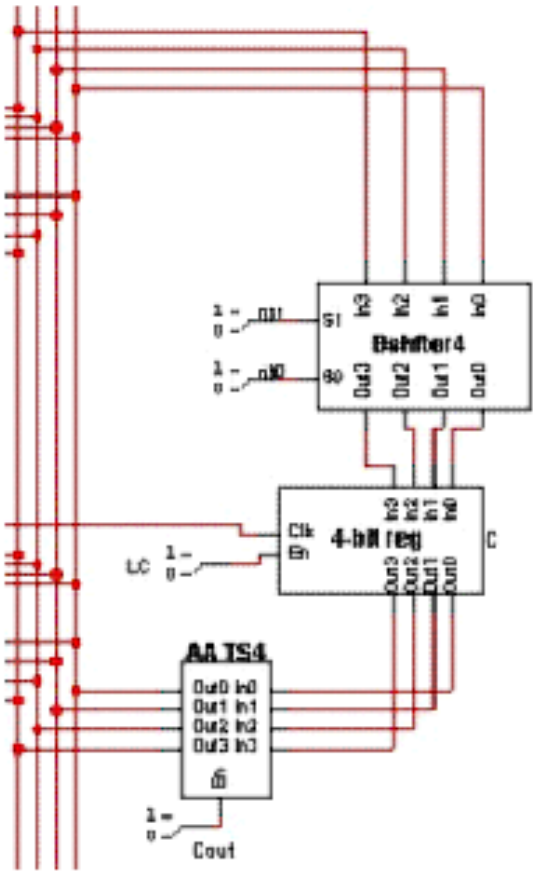
the first step is to activate R3out, nb1 and LC. Activating R3out will load the contents of the register R3 onto the bus. Since the bus is directly connected to the input of the barrel shifter, this number is applied to the input side. nb1 and nb0 are the barrel shifter's control lines for specifying the number of shifts to be applied. Applying a high input to nb1 and a low input to nb0 will shift the number two places to the right. Activating LC will load the shifted output of the barrel shifter into the



Barrel Shifter Symbol

S1	S0	Output in terms of the inputs
0	0	In3 In2 In1 In0
0	1	In0 In3 In2 In1
1	0	In1 In0 In3 In2
1	1	In2 In1 In0 In3

Function table: Barrel shifter



Shift operation using Barrel Shifter

register C. The second step is to transfer the contents of the register C to the register R4. This is done by activating the control Cout, which will load the contents of register C onto the data bus, and by activating the control LR4, which will let the contents of the bus be written to the register R4. This will complete the conditional shift-and-store operation. These steps are summarized in the table shown below.

Time step	Operation to be performed (structural RTL)	Control signals to be activated
1	C ← R3 (after rotating right twice)	R3out, nb1, LC
2	R4 ← C	LR4, Cout

Structural RTL: Shift operation