Realization of Verilog HDL Computation Model

CSCI 320 Computer Architecture

By Dr. Daniel C. Hyde
Department of Computer Science
Bucknell University
October 1997

1. Introduction

Verilog HDL is a hardware description language. The computer industry uses Verilog HDL for design of digital systems. From the Verilog code, industry uses automated tools to generate integrated circuit masks needed in the fabrication of integrated circuit (IC) chips.

In this course, a subset of the Verilog language will be used to describe and develop computer architectural concepts using a model of computation at the register transfer level. The purpose of this handout is to describe this model of computation and demonstrate how the model can be realized in digital circuits.

2. Example Simple Computer Using Register Transfer Level

Assume we have a very simple computer, with 1024 words of memory (**Mem**) each 32-bits wide, a memory address register (**MA**), a memory data register (**MD**), an accumulator (**AC**), an instruction register (**IR**) and a program counter (**PC**). Since we have 1024 words of memory, the **PC** and **MA** need to be 10-bits wide.

We can declare the registers and memory in Verilog as the following:

```
reg [0:31] AC, MD, IR;
reg [0:9]  MA, PC;
reg [0:31] Mem [0:1023];
```

We assume a digital system can be viewed as moving vectors of bits between registers. This level of abstraction is called the *register transfer level*. For example, we may describe in Verilog an *instruction fetch* by four register transfers:

```
// instruction fetch
#1 MA <= PC;
#1 MD <= Mem[MA];    // memory read
#1 IR <= MD;
#1 PC <= PC + 1;
```

The meaning of the first line is to transfer the 10 bits of the **PC** into the **MA** register *after* waiting one clock period (the **#1**). Note that it is important that we use Verilog's *blocking* assignment operator (<=) rather than the *non-blocking* assignment (=). The reason why will be discussed later.

Assuming the operation code for a LOAD instruction is 100 in binary and is in the first three bits of **IR**, we could design the *decode* and *execute* part of a LOAD instruction as follows:

```
// decode and execute code for a LOAD
#1 if (IR[0:2] == 3'b100) begin
     #1 MA <= IR[22:31]; // last 10 bits are address
     #1 MD <= Mem[MA];   // memory read
     #1 AC <= MD;
end
```

We will use Verilog to <u>describe</u> our digital systems but also to <u>test</u> our <u>designs</u> by a *simulator* running on our Unix workstations. The **#1** at the beginning of the Verilog statements means wait one unit of *simulation time*. Verilog calls the feature "delay control." The above design is very slow! A LOAD instruction will take eight clock periods. Later we will carefully analyze the Verilog code to explore ways to improve the speed.

3. Our Computational Model

Register transfers are general. In fact, we can view a computation as a specific class of register transfers.

Definition: A **computation** consists of placing some *Boolean* function of the contents of argument registers into a destination register.

That is, in this course a computation is a register transfer. In computer design, register transfers are very important. They are everywhere -- in arithmetic logic units (ALU), control units (CU), memory subsystems, I/O devices and interconnection networks.

Observe from the above definition that we need only Boolean functions. We don't need arithmetic or higher order functions. To realize the Boolean functions, we design combinational logic circuits, for example, an adder, from AND, OR and NOT gates.

Boolean functions have no concept of time. To incorporate the passage of time in our model, we define computing as a sequence of several register transfers where each transfer takes one clock period.

Definition: Computing is a sequence of computations.

Therefore, the above Verilog code for the LOAD instruction has seven computations or register transfers. We would say the performing of a LOAD instruction is computing because we do seven computations one after the other, in sequential order.

A major part of the description of a computer is a plan defining each register transfer, or computation, and specifying the order and timing in which these register transfers will take place.

How does this view of computation compare with other definitions you may have seen? One way to view computation is as a change in *state*. Here in a very practical way, we are viewing computation as a change in the state of hardware registers. For example, the MA changes state in the following transfer.

```
#1 MA <= PC;
```

One of the clever parts of the Verilog language design was making register transfers look like assignment statements in other programming languages. Since many designers are comfortable with a language like C or Pascal, Verilog has had a large degree of success.

Other parts of the Verilog language allow the designer great flexibility to create starting with pass transistors a hierarchy of combinational logic functions to be used in register transfers (called *structural* level in Verilog).

It should be noted that we are interested in describing *digital systems*. There is a whole world of analog or continuous systems, e. g., FM tuners, in which our definition of computation ignores.

4. Realization of Verilog Code into Digital Logic Circuits

Our goal is to demonstrate that our subset of Verilog code can be realized in digital logic circuits. Also, we will show that this translation from Verilog code to digital circuit can be automated. It is this use of automated design tools that has allowed the computer industry to create extremely complex and sophisticated designs such as Intel's Pentium chip.

A register is a series of flip flops. We will assume for our model that all flip flops are *trailing edge triggered* clocked **D** flip flops. Therefore, the declaration of the **A** register in Verilog:

```
reg [0:3] A;
```

means the four flip flops as shown:

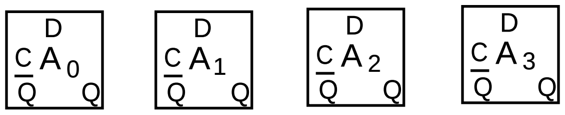

Fig. 1 Register **A** is four **D** flip flops.

A computation is a register transfer. Here are two computations and their associated declaration.

```
reg [0:3] A, B;

#1 A <= 4'b1010;  // set register A to 1010
#1 B <= A;
```

The second register transfer is modeled by the following diagram:

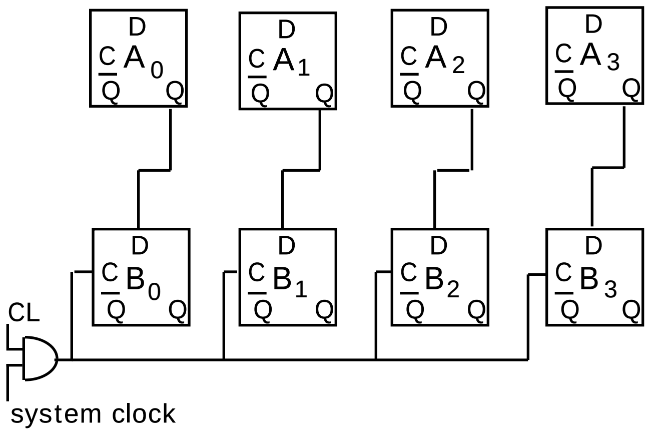

Fig. 2 Data unit for register transfer B<= A.

The transfer is only done when the control signal **CL** is 1 and system clock goes from 1 to 0 (assuming trailing edge triggered). Our Verilog notation models the situation because the blocking assignment operator (<=) means to block the assignment until the end of the current unit in simulation time.

Where does the control signal **CL** come from? From the control unit which sequences the register transfers. Any digital system has two parts: the *data unit* (also called *data path*) and the *control unit* as shown below:

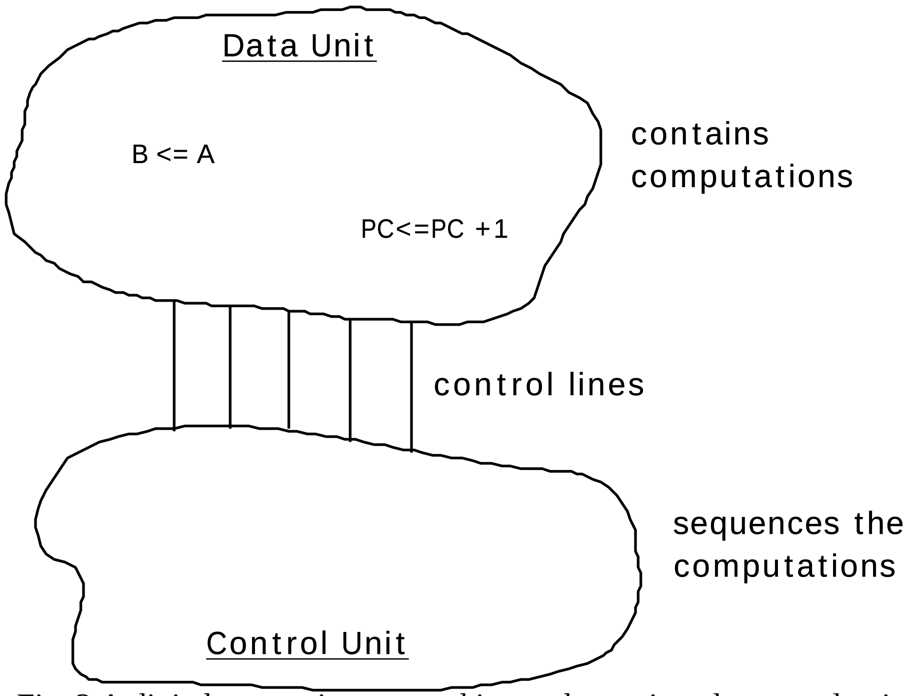

Fig. 3 A digital system is separated into a data unit and a control unit.

We will design control units in a later section.

The above diagram of a register transfer is misleading, register transfers are not that simple. Typically, we transfer from several argument registers to one destination register. Given the following register transfers used somewhere in the *control sequence*:

```
reg [0:3] A, B, X;

A <= B    // uses control level CL1
A <= X    // uses control level CL2
```

here is the data unit for the two computations:

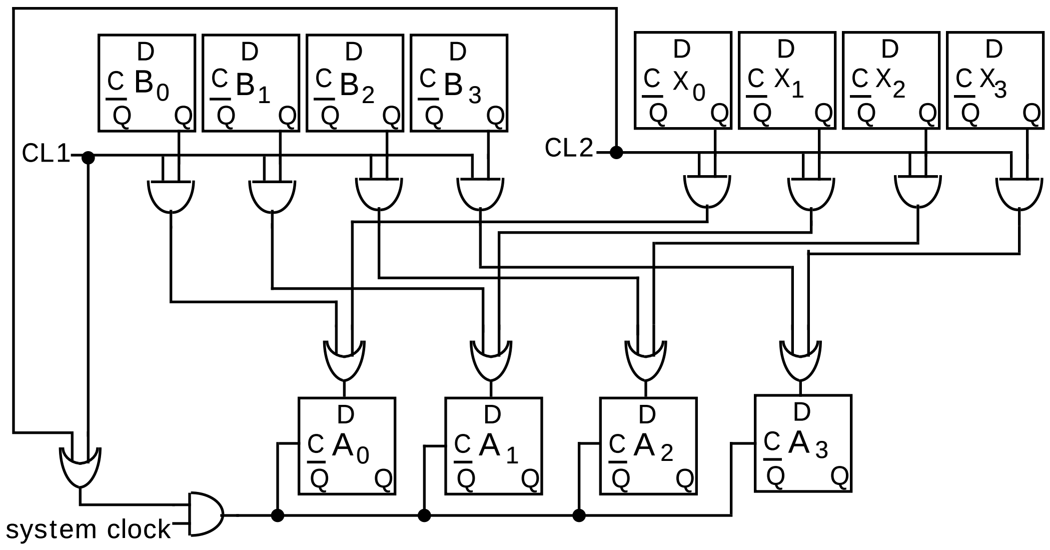

Fig. 4 The data unit for A <= B and A <= X.

Extra AND and OR gates are needed to gate the proper bit values into the **D** input of the **A** flip flops.

A bank of ANDs is needed for each argument register and a bank of ORs for the input into the **D** input of the destination flip flops. A technique called "busing" would reduce the number of AND and OR gates needed. However, we will ignore busing.

Observe that the data unit does *not* contain when the computations are done. The sequencing of the three is done by the two control signals supplied by the control unit.

Here is another example of a computation and how we model the Verilog code as a digital logic circuit:

```
reg [0:3] A, B, X;

A <= X | ~B  // | is bit-wise OR and
             // ~ is bit-wise NOT (complement)
```

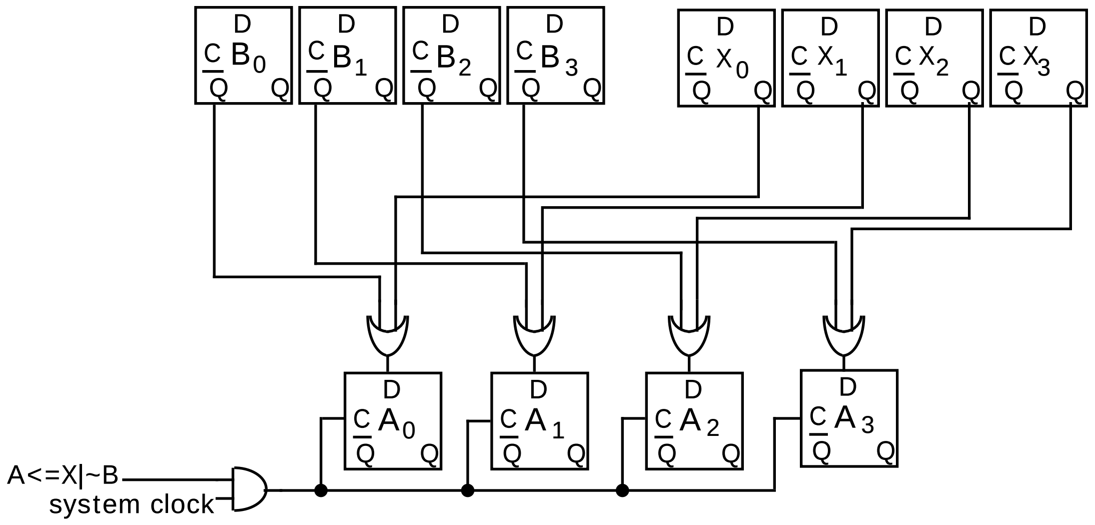

Fig. 5 The data unit for A <= X | ~B.

Observe that we use the NOT **Q** output of the **B** flip flops. Also, note the convention where we label the control signal with the character string of the computation. As digital systems become more complicated, this convention helps the reader sort out what is happening.

Here is a circular right shift of register A.

A <= {A[3], A[0:2]};

The { } braces and comma are for concatenation of bit strings in Verilog. Here is the corresponding diagram.

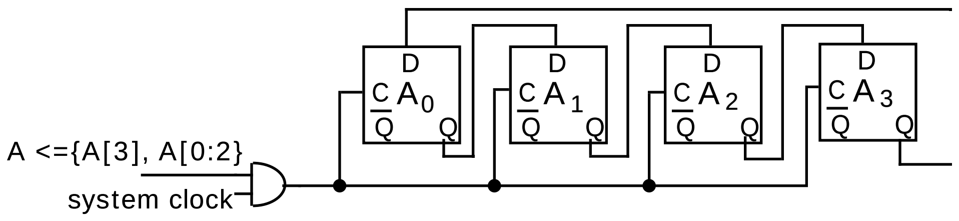

Fig. 6 The data unit for circular right shift of register A.

What else needs to be realized? We need to reduce all operations to combinational circuits (ones composed of only AND, OR and NOT gates). For example, we need to realize arithmetic, for example, the “+” operator by an ADDER circuit. Usually, we also design special combinational circuits for increment by one (INC) and decrement by one (DEC). For example, we would replace the + operator used in the instruction fetch

```
#1 PC <= PC + 1;
```

with

```
#1 PC <= INC(PC);
```

We need to design combinational circuits for comparison operators, such as `= =`. For example, the following

```
#1 if (IR[0:3] == 4’b1010) begin
```

would be replaced with

```
#1 if (EQUAL(IR[0:3], 4’b1010) ) begin
```

The design of the combinational logic functions INC, DEC, EQUAL and GREATER are left as exercises for the reader.

One advantage of Verilog is that it does not require the designer to reduce every operator to combinational functions (Verilog’s structural level). If appropriate, say at the beginning of a design, Verilog allows the designer to use familiar arithmetic operators such as `+`, `= =` and `>` (Verilog’s behavioral level). Verilog allows an incremental design strategy which is especially important for learning.

It should be noted that we also need to explore how we model memory. Our read of memory in the above instruction fetch assumes a synchronous memory which can be read in one clock period, This may not be realistic in the system we are designing.

5. Designing the Control Unit

Now that we know how to realize the data unit with its computations, we want to be able to sequence the computations in the proper order. A control unit does the sequencing. Note: a digital system might have several control units. We will discuss the design of a *hardwired control unit.* A second technique used to design control units is *microprogramming*. Microprogramming though not difficult is beyond the scope of the handout.

Given the following Verilog control sequence:

```
initial begin
     #1 A <= B;      // no. 1
     #1 C <= D;// no. 2
     #1 A <= ~B;     // no. 3
end
```

the control unit would place a 1 on the control line for **no. 1** for a clock period. Then a 1 on control line for **no. 2** for a clock period. Then a 1 on control line for **no. 3** for a clock period. That is, do 1, do 2, do 3, halt. Observe that the control unit does *not* know nor care what each computation does. We assume here that a computation takes one clock period.

Our control unit starts with a single (1) pulse (a clock period wide) from a *single pulse generator*.

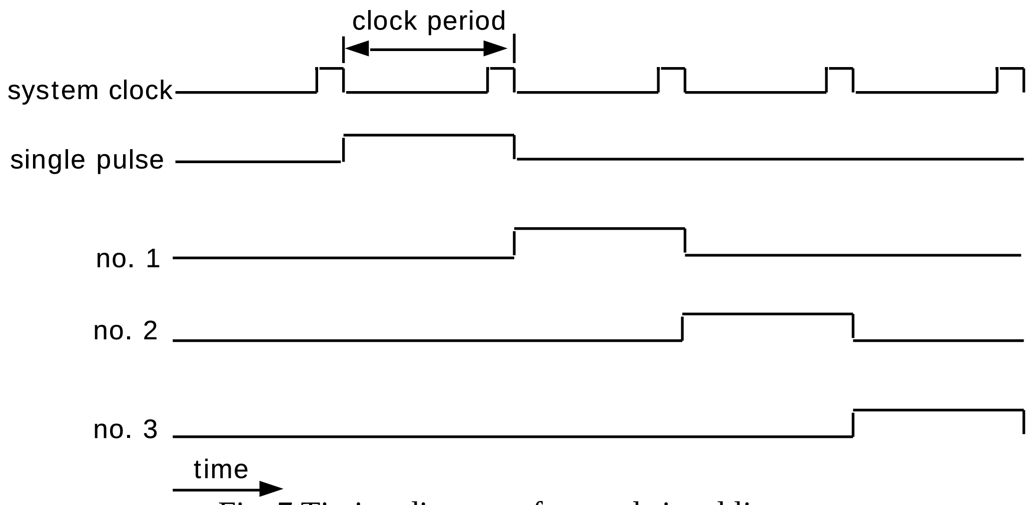

Fig. 7 Timing diagram of control signal lines.

Notice that the pulse, called the control pulse, is delayed one clock period as it raises the control lines for the three computations. What kind of circuit can delay a pulse for one clock period? We can use our old friends the trailing edge triggered clocked **D** flip flops.

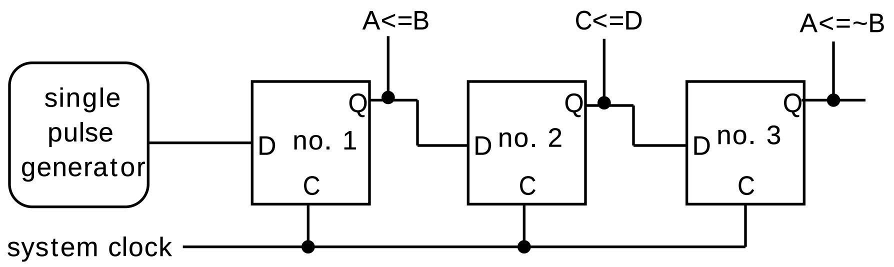

Fig. 8 Hardwired control unit.

Rules for construction of hardwired control unit

1. Label each Verilog statement which starts with **#1** (delay control).
2. Use one **D** flip flop for each labeled statement. Label flip flop with same label.
3. If more than one input to the **D** input of a flip flop, insert an OR gate.
4. *More Rules to come.*

Here is a control sequence with an **always** construct instead of an **initial**. The **always** construct in Verilog is semantically the same as an **initial** except it repeats the control sequence forever.

```
always begin
     #1 A <= B;     // no. 1
     #1 C <= D;// no. 2
     #1 A <= ~B;    // no. 3
end
```

Here is the control unit. Notice that we need to apply Rule 3 and insert an OR gate. The way to think about this control unit is that the single control pulse travels around and around forever.

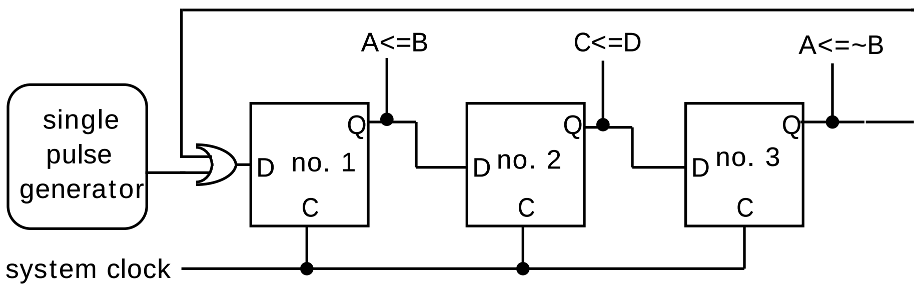

Fig. 9 Hardwired control unit for an **always** construct.

Verilog is a *structured programming* language in the tradition of Pascal and C, that is, the flow of control is by **while** and **if-then-else** constructs and not **goto** statements.

For example, below is a **while** loop:

```
while (s)
   begin
      #1 A <= B;      // no. 1
      #1 C <= D;// no. 2
      #1 A <= ~B;     // no. 3
   end
```

where **s** is a logical expression, i. e., results in a 0 or 1. We will assume that this **while** loop is part of a larger control sequence of an **initial** or **always** construct. Therefore, we will show the **in** and **out** for the control pulse. Notice we need a new rule -- Rule 4. (Note: the designer might need a **#1** in front of the **while** if any value in **s** uses a result set just before. Called a *data dependency*. See section Speeding Up the Machine.)

Rule 4: If a conditional construct, i. e., a **while** or an **if**, send control pulse both directions depending on AND gates with **s** and NOT **s** as inputs.

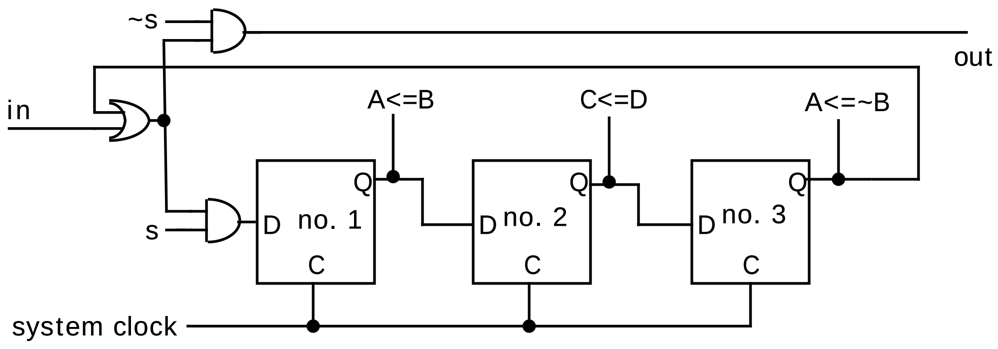

Fig. 10 Portion of a hardwired control unit for a **while** construct.

Example **if** construct:

```
if (s)
   begin
      #1 A <= B;  // L1
      #1 C <= D;  // L2
   end
else
   begin
      #1 D <= A;  // L3
      #1 X <= Y;  // L4
```

```
        end
```

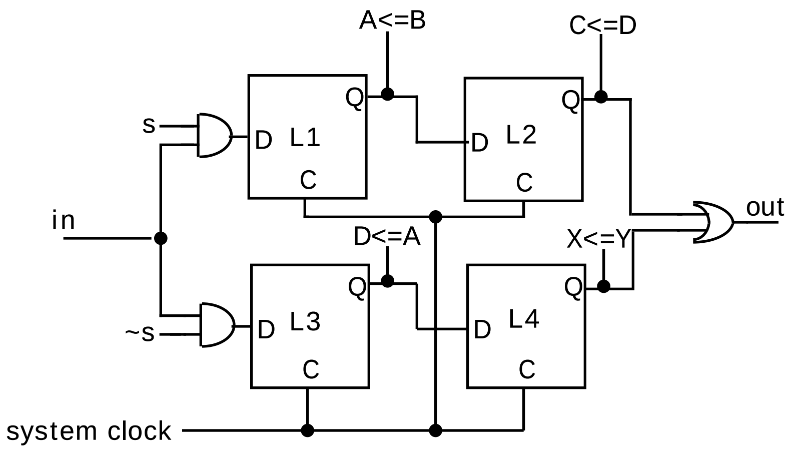

Fig. 11 Portion of hardwired control unit for an **if** construct.

Again we need a new rule. Note that the pulse goes one way or the other, never both.

Rule 5: For an **if** construct, we need an OR gate to combine the two possible paths of the control pulse.

Theorem: **Sequence**, **if-then-else** and **while** are enough control of flow constructs to do *any* algorithm. (Proven by Jacoppini and Bohm in 1966)

Because of the above theorem, we have all the control constructs we need. However, for designer convenience, Verilog has a **case** construct, a **for** and a **repeat**. Handling their translation to digital logic circuits is straight forward since any **case** can be transformed into **if**s and any **for** or **repeat** can be transformed into **while**s.

Another important control construct in Verilog is the **fork-join**. The **fork-join** introduces *concurrency* into the control sequence, i. e., *multithreads of control*. The reader should recall that all Verilog **initial** and **always** constructs are already performed concurrently (conceptually each has its own control unit). The **fork-join** is concurrency within a single **initial** or **always** construct.

```
fork
     #1 A <= B;
     #1 Q <= P;
     #1 C <= D;
join
```

The **fork** sends the control pulse **n** ways, one for each statement inside (can be **begin-end** constructs).

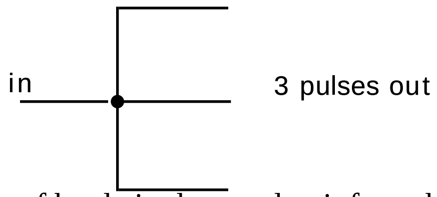

Fig. 12 Portion of hardwired control unit for a three-way **fork**.

Rule 6: For a **fork**, the control pulse is sent **n** ways.

The **join** synchronizes the control pulse after all **n** pulses have arrived. This is performed by a special circuit.

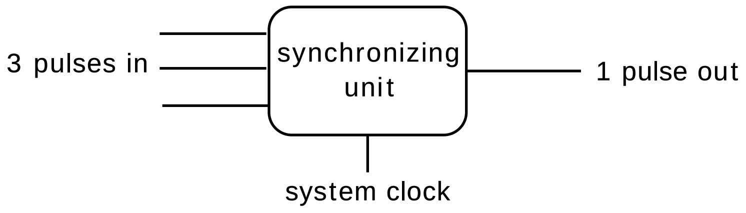

Fig. 13 Portion of the hardware control unit for a three-way **join**.

Rule 7: For a **join, the n** control pulses are synchronized by a special circuit which passes along one control pulse after all **n** pulses have arrived.

This concludes the design of the hardwired control unit.

6. Speeding Up the Machine

Fast performance is many times important, especially with CPUs. Many times, register transfers can be performed in the same clock period. For example, given the following register transfers:

```
#1 A <= B;  // first
#1 C <= D;  // second
```

we can remove the second **#1** and have the transfers done in the same clock period.

```
#1 A <= B;
   C <= D;
```

The semantics of the Verilog *blocking assignment* says to evaluate the right-hand sides and block all the assignments to left-hand sides until the end of the current unit of simulation time. In our control unit, we would eliminate the second **D** flip flop and the control lines for the two register transfers would originate from the same place, i. e., the **Q** output of the first **D** flip flop.

We can only remove **#1**s when there is no *interference* between registers, e. g., transferring to same register, and no *data dependencies*, i. e., the results of one transfer is needed as argument to later transfer. For example, a data dependency exist in the following:

```
#1 MA <= PC;
#1 MD <= Mem[MA];
```

because the results of the first transfer is needed in the second. In this case, one cannot remove the **#1**. If one did, the MA in the second transfer would be the old value of the **MA**. However, one can do this:

```
#1 MA <= PC; PC <= PC + 1;
```

because both transfers use the old value of the **PC**. Remember with the Verilog blocking assignment (<=), all the assignments to the left-hand sides are blocked until the end of the current unit of simulation time.

We need to be careful with logical expressions in **while**s and **if**s because of possible data dependencies. For example,

```
#1 IR <= MD;
if (IR[0:2] == 3'b100) begin
```

is probably not what was intended. The **if** statement compares the old value of the **IR** not the value set in the previous line. Here we need to add a **#1** in front of the **if** statement.

Usually, we write our design using **#1**s liberally. After we have tested the design and are confident that it works, then we carefully analyze the code and remove the **#1**s one at a time.

7. Conclusions

Given a complicated digital system description in Verilog, translating the Verilog code to the digital circuit becomes very tedious and time consuming. Hence, the utility of automated tools to perform this task. From the few examples shown, you should begin to see how the task of translating Verilog non-blocking assignment statements, e. g., **A<= B**, to data unit could be automated. Similarly, you should be able to see how to create the control unit circuits from the Verilog control sequence.

It is important to understand that our goal is to realize our Verilog code in digital circuits so we no longer need to think in digital circuits. That is, we are developing a higher level of abstraction to think about digital systems that are much more concise than digital circuits, e. g., sequential machines. A few lines of Verilog code may translate into hundreds of flip flops, AND, OR and NOT gates. This Verilog model is precise and concise -- the Verilog notation supplies the information for *both* the <u>data unit</u> and the <u>control unit</u> associated with the control sequence. This approach is basically the one used in industry to design digital integrated circuits, such as CPU chips. With automated tools, the Verilog code is translated to the integrated circuit masks, say, for CMOS with p-channel and n-channel transistors.