RaPiD-C Manual[1]

Carl Ebeling
Department of Computer Science and Engineering
University of Washington
August 20, 2002

UW CSE Technical Report UW-CSE-02-07-06

Abstract

This manual describes the RaPiD-C language and shows how it is used to implement computations using the RaPiD datapath. It is assumed that the reader is familiar with the RaPiD architecture and in particular with the structure of the RaPiD datapath. The most important feature of the RaPiD datapath is that it is a collection of functional units such as ALUs, multipliers, shifters, memories and registers, which are configured into a linear datapath that executes a computation. A RaPiD-C program describes this computation by explicitly specifying the computation performed by each functional unit on each cycle. The goal of the RaPiD architecture is to execute repetitive, compute-intensive algorithms by performing many operations in parallel. The programmer typically views the RaPiD datapath as a set of processing elements, each of which performs a computation on each clock cycle. These processing elements are called *stages* in RaPiD because the data usually flows from one stage to the next in a pipelined way to form a pipelined computation. The number of stages, the computation performed in each stage, and the data communication between stages is described by a RaPiD-C program.

Broadcast Computation Model and Datapath Instructions

A key feature of the RaPiD-C programming language is the broadcast model used for communication. That is, RaPiD-C allows the programmer to think about the computation in a non-pipelined manner where data can travel from one end of the datapath to the other during a single cycle. For example, input data can be broadcast to all the stages in one cycle, or output data can be accumulated across all stages and sent to an output stream in one cycle. A computation is pipelined when it is executed, that is, there is a clock cycle delay in the movement of data from one stage to the next. But the compiler takes care of this pipelining, and the programmer does not need to think this way. The debugger also operates in the broadcast mode so the programmer can debug the program using the same model as he used to write it.

```
for (s = 0; s < STAGES; s++)
```

The basic instruction of a RaPiD-C program is the **`Datapath`** instruction which describes the computation performed during a single clock cycle by each stage of the datapath. The **`Datapath`** instruction is really just shorthand for the following loop which iterates the reserved variable **`s`** over all the stages:

```
Datapath {
   if (s==0) result = inData[s] * weight[s];
   else      result = result + inData[s] * weight[s];
}
```

That is, the statements in the Datapath instruction are executed for all values of **`s`** from 0 to **`STAGES-1`**, where **`STAGES`** is a constant defined by the user that specifies how many stages are used in the datapath. For example, the Datapath statement[2]:

performs the vector multiply of the two vectors **`inData`** and **`weight`**, producing the value **`result`**. This computation occurs in a single cycle in the RaPiD-C program, although when implemented it will be pipelined over several cycles. The executable statements in the Datapath statement are executed by every stage, where the reserved

[1] This research was supported by the National Science Foundation Experimental Systems Program (EIA-9901377).
[2] The early examples in this manual are not complete programs. Later on, when all the elements of a RaPiD-C program have been introduced, we will use complete programs as examples.

variable **s** is used to modify the computation performed by stage **s**. The notation **inData[s]** and **weight[s]** indicates that these two vectors are stored one element per stage. Stage 0 executes the then clause of the if statement, while the remaining stages execute the else clause. The variable **result** is a special variable that is used to accumulate the results of the multiply-accumulate operation executed by each stage. This Datapath statement compiles into the datapath shown in Figure 1.

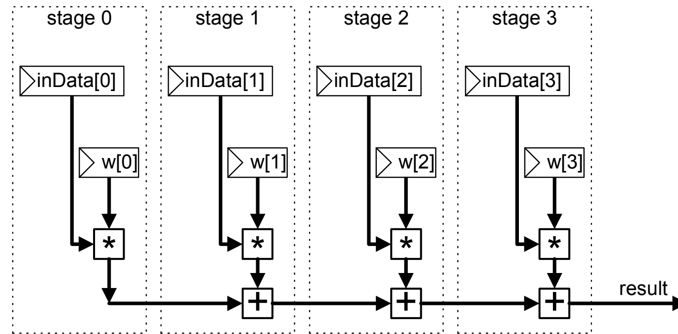

Figure 1 - Simple inner production computation

This example illustrates a very important feature of the RaPiD-C language. The computation specified in the **Datapath** statement is executed in a single cycle regardless of how many operations are specified. In this example, the entire loop of **s** from 0 to **STAGES-1** is executed, in order, in a single cycle. The variable **result** is in effect a temporary variable that is used repeatedly within this computation to produce the dataflow shown in Figure 1.

The broadcast model thus underlies how the programmer thinks about a computation, where Datapath instructions describe what each stage does during a single clock cycle, and data can be communicated from one end to the other of the datapath during that cycle. Note, however, that this broadcast communication goes in one direction, from the input streams on one end to the output streams at the other because of the order of the loop over **s**. Communication can go in the opposite direction, "upstream", but not within one clock cycle. This will be shown later when BackPipes are introduced.

Control Loops

The Datapath instruction describes what happens during one clock cycle but complete computations take many clock cycles. RaPiD applications usually are very repetitive, with the same or very similar computation performed repeatedly on data which changes from cycle to cycle. Thus a Datapath instruction is usually repeated many times. This is done in RaPiD-C using for loops, which are very similar to regular C for loops except that the for loop variable is a special type and must be declared at the beginning of the program. Moreover, each for loop must use a unique For variable.

So, for example, to execute the Datapath statement above N times, it would be put in a simple for loop:

```
For i;

for (i = 0; i < N; i++) {
    Datapath {
        if (s==0) result = inData[s] * weight[s];
        else      result = result + inData[s] * weight[s];
    }
}
```

Now, of course, this just does the same computation over and over again and so it isn't very interesting. But if we were able to change the data in the **inData** vector, we could perform a different vector multiply on each clock cycle. We'll see how to do this later on.

Placing the Datapath statement in a For loop does not change the dataflow graph that is compiled into the RaPiD datapath. The for loop simply specifies how many cycles to perform the computation described by the Datapath statement.

For loops can be nested in the usual way, but each loop must use a unique For loop variable. For loops can also appear sequentially: the Datapath statement in the first For loop is executed for a specified number of cycles, and then the Datapath statement in the second For loop is executed a specified number of cycles. The RaPiD datapath is configured so that it can execute either Datapath statement. The For loops are executed by a controller which sends control signals to the datapath to indicate which For loop is being executed and thus which Datapath statement should be executed.

For loop variables can also be used to modify the computation performed in the Datapath statement. Unlike the s variable, which has a constant value for each stage, which is known at compile time, For loop variables take on different values during the computation. For example, in the following program:

```
For i;

for (i = 0; i < N; i++) {
    Datapath {
        if (i==0) sum[s] = inData[s] * weight[s];
        else      sum[s] = sum[s] + inData[s] * weight[s];
    }
}
```

the local variable sum[s] in each stage is initialized when `i==0` and then added to thereafter. (This is not a particularly useful computation, since it just adds `inData[s] * weight[s]` in each stage N times.)

Here are some useful abbreviations that can make programs more readable. Given the loop:

```
for (i=0; i<N; i++)
    i.first:     True during the first iteration, ie. i==0
    i.last :     True during the last iteration, ie. i==N-1
    i.live :     True during the scope of the loop
```

The `i.first` and `i.last` abbreviations are obvious; the `i.live` notation can be useful when writing parallel threads.

Data and Variables

The datapath operates on data which is defined by the specific implementation of the RaPiD datapath. For example, the simulator currently supports 16-bit fixed-point data. The data type Word is used to declare a variables used in the computations described by Datapath statements. Arrays of Words are used for declaring data vectors whose elements are assigned one per stage. For example,

```
Word x[STAGES];
```

declares a vector `x` with `STAGES` elements, one per stage. A Datapath statement could add all the elements of `x` as follows, again using a special variable `result`:

```
Word x[STAGES];

Datapath {
    if (s==0) result = x[0];
    else      result = result + x[s];
}
```

(Recall that the statements in the Datapath instruction are executed by all the stages, from 0 to `STAGES-1`.)

Variables in RaPiD-C are used to describe a dataflow computation. This computation occurs both within a clock cycle and across clock cycles. In the following computation:

```
For i;
Word sum[STAGES], inData[STAGES], weight[STAGES];
Word temp[STAGES];
    for (i = 0; i < N; i++) {
       Datapath {
           temp[s] = inData[s] * weight[s];
           if (i==0) sum[s] = temp[s];
           else      sum[s] = sum[s] + temp[s];
       }
    }
```

the variable **temp[s]** is used only to describe the computation within one clock cycle. That is, its value is not carried from one clock cycle to the next and thus does not need a register to store it. Variables that carry values from one cycle to the next must of course be stored in a register. In this example, **sum[s]** is accumulated over the course of the loop and must be stored in a register that maintains its value from one clock cycle to the next. The compiler figures out where registers are needed and the programmer generally doesn't care where registers are assigned.

The programmer should be able to write this computation without the temporary variable, as we did previously. The compiler should recognize the common subexpression and assign a single multiplier to this computation.

Note that the execution model of RaPiD-C uses sequential assignment semantics. That is, statements are executed in the order that they appear. There is no "delayed assignment" like in Verilog. This fact becomes important when we write parallel threads that operate on shared values.

Streams

Data enters and exits the datapath via input and output streams. The simplest way to access external data is to simply read or write directly using a stream variable, declared and used as in the following example, which simply copies data from one stream to another:

```
StreamIn strInput;
StreamOut strOutput;
Pipe inData;
For i;

for (i = 0; i < N; i++) {
   Datapath {
       if (s==0) inData = strInput;
       if (s==STAGES-1) strOutput = inData;
   }
}
```

This program does not specify how the data gets into the input stream or where the data in the output stream goes. In some cases, streams are written or read directly by an external process. The RaPiD-C program can also specify how data in external memory is read into input streams or written to output streams, which is shown at the end of this section. (The special Pipe variable inData will be explained in detail in the next section.)

If the external data resides in external memory, the RaPiD-C program can reference this memory explicitly as an array reference, and the compiler will allocate a stream through which the data is read or written. For example, we can change the previous program to reference data in external memory in the arrays X and Y. The program now copies the array X to the array Y:

```
Pipe inData;
For i;
for (i = 0; i < N; i++) {
    Datapath {
        if (s==0) inData = X[i];
        if (s==STAGES-1) Y[i] = inData;
    }
}
```

Currently the compiler allocates a different stream to each reference to external memory, which is too expensive. The emulator, for example, is limited to 3 input streams and 1 output stream, or 2 input and 2 output streams. For example, the user cannot read two different arrays using the same stream as might occur when first reading in coefficient values followed by input data. Although for most programs the explicit array reference is convenient, it is often necessary to decouple the Stream definition and the Stream access, as shown in the following program. Note that the Stream definitions must go in separate parallel threads (which are presented later). These threads count against the maximum of 4 threads, unless virtual threads (vthread) are used.

Note that when explicit streams are used, the stream variable is the target of the assignment for both input and output streams.

```
StreamIn strInput;
StreamOut strOutput;
Pipe inData;
For i, j, k;

Par {
thread:
    for (i = 0; i < N; i++) {
        Datapath {
            if (s==0) inData = strInput;
            if (s==STAGES-1) strOutput = inData;
        }
    }
vthread: // This thread just defines the StrInput Stream
    for (j = 0; j < N; j++) {
        Datapath {
            StrInput = X[j];
        }
    }
vthread: // This thread just defines the StrOutput Stream
    for (k = 0; k < N; k++) {
        Datapath {
            StrOutput = Y[k];  // Note that the stream is target of assign
        }
    }
}
```

Pipes

Data moves through the datapath from one stage to the next via Pipes, which are special variables that provide a specific mechanism for data communication between stages. Variables declared as Words are localized to a single stage and cannot be referenced directly by another stage. For example, if `x[s]` is written in stage `s` then it cannot be read in stage s-1 or s+1. It would be illegal, for example, to say:

```
y[s+1] = x[s];
```

Instead, pipe variables must be used to pass data values between stages.

```
Word inData[STAGES], weight[STAGES];
For i;
Word result[STAGES];

for (i = 0; i < N; i++) {
    Datapath {
        if (s>0)  result[s] = result[s-1];
        // Initialization
        if (s==0) result[s] = inData[s] * weight[s];
        else      result[s] = result[s] + inData[s] * weight[s];
    }
}
```

A Pipe variable is like a Word variable except that it declares a variable in every stage. This Pipe variable in each stage is initialized at the beginning of every clock cycle with the last value written to the Pipe variable in the previous stage. The execution model for Pipes is given by the following example program. Note that this does not work in RaPiD-C because references across stage boundaries are not allowed.

Declaring **`result`** as a Pipe variable creates in effect a vector with one element per stage as above. The compiler inserts the first statement of the loop, which copies the value of the Pipe variable from the previous stage to the current stage. (This initial value is obviously undefined in the first stage.)

Since the value of the Pipe variable is copied from one stage to the next within the same clock cycle, it is clear that we need only one variable here. Thus the loop is actually written as follows in RaPiD-C, without the first initialization statement, which is executed implicitly, and with **`result`** as a simple variable instead of a vector.

```
Word inData[STAGES], weight[STAGES];
For i;
Pipe result;

for (i = 0; i < N; i++) {
    Datapath {
        if (s==0) result = inData[s] * weight[s];
        else      result = result + inData[s] * weight[s];
    }
}
```

Note that if the Pipe variable is not written in a stage, its value is passed along unchanged to the next stage. If no stage writes a value to the Pipe variable, it ripples down the pipeline from stage 0 to stage **`STAGE-1`**.

For example, the program:

```
Pipe inData;
For i;
for (i = 0; i < N; i++) {
    Datapath {
        if (s==0) inData = X[i];
        if (s==STAGES-1) Y[i] = inData;
    }
}
```

assigns the Pipe variable **`inData`** to an input value in the first stage. (**`X[i]`** is a reference to external memory which occurs via a compiler-generated input stream.) Since no other stage writes a value to **`inData`**, the value written to **`inData`** by the first stage is passed along from stage to stage and is finally available at the last stage. The dataflow for this Datapath statement is simply a wire from the first stage to the last carrying this value. That is, the input value is broadcast to the entire array via the **`inData`** Pipe variable. In the last stage, this value is then written to the output. (**`Y[i]`** refers to external memory and the output is done via an output stream.)

In summary, a simple Pipe variable can be viewed as a global variable that can be read and written by every stage in a single clock cycle as the stage variable **`s`** iterates from 0 to **`STAGES-1`**.

Pipe variables can also be declared with a delay value. A Pipe variable with a delay of 1 is different from a Pipe variable with no delay in that it is initialized to the last value written to the Pipe variable of the previous stage *in the*

previous clock cycle. That is, the Pipe variable value moves from one stage to the next, delayed by a clock cycle between stages. This is implemented by inserting a register between each stage. At the end of a clock cycle, the last value of the Pipe variable *in each stage* is written to this register and this value is then used to initialize the Pipe variable in the following stage in the next clock cycle. This register is required for a delayed Pipe variable because the values are saved from one clock to the next. No register is needed for a 0-delay Pipe variable because the value is used only within a single clock cycle. The model for delayed Pipe variables is shown in Figure 2. Pipe variables can be declared with any delay k, in which case k registers are inserted between each stage. However, a delay greater than 1 is very seldom used.

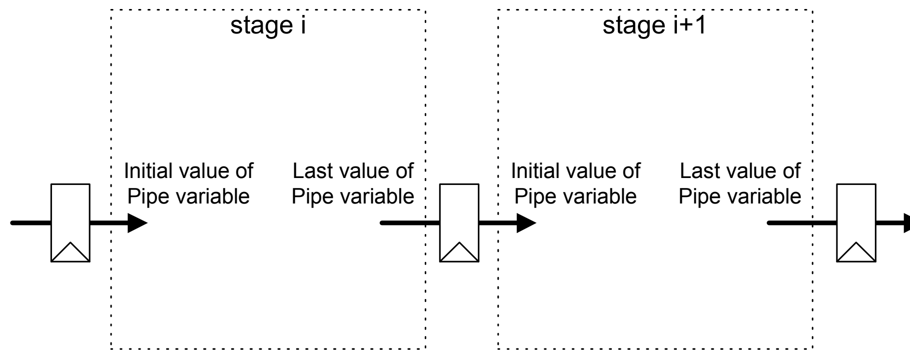

Figure 2 - Registers are used to implement delayed Pipe variables

In the case of delayed Pipes, the Pipe variable can be viewed as a vector since the values written by each stage are held until the next cycle, at which point the values are shifted down the datapath one stage and are then referenced by the following stage.

In the following example, the **`inData`** variable is a Pipe variable with a delay of 1.

```
Word weight[STAGES];
Pipe result, inData(1);
For i;
for (i = 0; i < N; i++) {
    Datapath {
        if (s==0) {
            inData = X[i];
            result = 0;
        }
        result = result + inData * weight[s];
    }
}
```

In this example, **`result`** is a Pipe variable with delay 0, that is a simple “broadcast” variable which is available in each stage in turn in one cycle. **`inData`** is a Pipe variable with delay 1, which means that it is in fact a vector which is shifted down the datapath one stage per cycle. This program now does something interesting, although the result is not saved anywhere. On each cycle, the **`weight`** vector is multiplied with the Pipe vector **`inData`**, but now this **`inData`** vector is changed on every cycle by shifting the input data through it at a rate of one value per cycle. This is the simple FIR (Finite Impluse Response) filter computation.

Currently in RaPiD-C, all Pipe variables are referenced as simple variables even though delayed Pipes are really vectors. However, the programmer thinks of delayed Pipe variables as vectors.[3]

---

[3] I would prefer to think of Pipes with delay as Pipe vectors (arrays) and I would force the programmer to refer to these Pipes vectors explicitly as vectors. Programs would be clearer this way. Declaring a Pipe enables assignments across stages and causes the assignment **`inData[i] = inData[i-1]`** to be executed by default. Of course, there are cases where the programmer doesn't want this assignment executed every cycle. So we should probably define a different kind of Pipe variable that doesn't shift automagically.

```
Word weight[STAGES];
Pipe InData[STAGES];            // InData is really a vector
Pipe result;
For i;
for (i = 0; i < N; i++) {
   Datapath {
      if (s==0) {
         inData[0] = X[i];
         result = 0;
      }
      result = result + inData[s] * weight[s];
   }
}
```

A Pipe variable with a delay of 1 implicitly implements a shift register that shifts data through the elements of the Pipe vector one stage per clock cycle. For example, the following program copies data from an input stream to an output stream but now the data is shifted down the array, taking **`STAGES-1`** cycles to make it from input to output:

```
Pipe InData(1);
For i;
for (i = 0; i < N; i++) {
   Datapath {
      if (s==0) inData = X[i];
      if (s==STAGES-1) Y[i] = inData;
   }
}
```

This program does the wrong thing of course, since the **`inData`** Pipe vector starts out "empty". It takes **`STAGES-1`** cycles before the first data entered makes its way to the last stage and out of the datapath. The solution is to fill the **`inData`** vector before writing data to the output stream. Another problem is that when the input data is exhausted, the datapath is still full of data that needs to be sent to the output. The following program solves this by adding an initialization and finalization section.

```
Pipe InData(1);
For init, i, final;
for (init = 0; init < STAGES-1; init++) {
   Datapath {
      if (s==0) inData = X[init];
   }
}
for (i = 0; i < N - (STAGES-1); i++) {
   Datapath {
      if (s==0) inData = X[i+STAGES-1];
      if (s==STAGES-1) Y[i] = inData;
   }
}
for (final = 0; final < STAGES-1; final++) {
   Datapath {
      if (s==STAGES-1) Y[final+N-(STAGES-1)] = inData;
   }
}
```

Clearly, it is very easy to be off by one when programming Pipes like this. The key is that there are actually **`STAGE-1`** registers in the Pipe, one between each pair of neighboring stages. Thus it takes **`STAGE-1`** iterations to fill the Pipe vector, after which values can be written to the output stream. After the last input value has been read, there are still **`STAGE-1`** values in the vector and it thus takes **`STAGE-1`** cycles to empty the vector. For example, if there are 2 stages in the datapath, one cycle is used to read a data value into the first stage. On the next cycle, the value is available in the second stage where it can be written to the output stream.

Of course, this could have been written using a single loop, like this:

```
Pipe InData(1);
For i;
for (i = 0; i < N + (STAGES-1); i++) {
    Datapath {
        if (s==0 && i < N) inData = X[i];
        if (s==STAGES-1 && i >= STAGES-1) Y[i-(STAGES-1)] = inData;
    }
}
```

Neither of these ways of writing this program are particularly straightforward. We will show a better way to write this program later on using parallel threads.

Explicit and Enabled Pipes

Pipe variables must be used when moving data from one stage to the next. They provide a convenient, if perhaps confusing, way to move data automatically down a pipeline for most applications. There are times when a more general mechanism is necessary. For example, Pipes are defined to shift data on every cycle but sometimes the program needs to shift only when necessary. The program example **doublePipe** at the end of this document shows how this is done.

Memories

The RaPiD datapath contains a set of memories that can be used to store and reuse data in the datapath. These memories contain a small number of entries (64-256) and are addressed using a special incrementing address register. These memories are declared explicitly as type RAM. A RAM's address register supplies the address that is used to both read and then write an entry in the RAM on the same cycle. This address register can be cleared, incremented and loaded from a value in the datapath. Figure 3 shows the structure of the RAM with the address register.

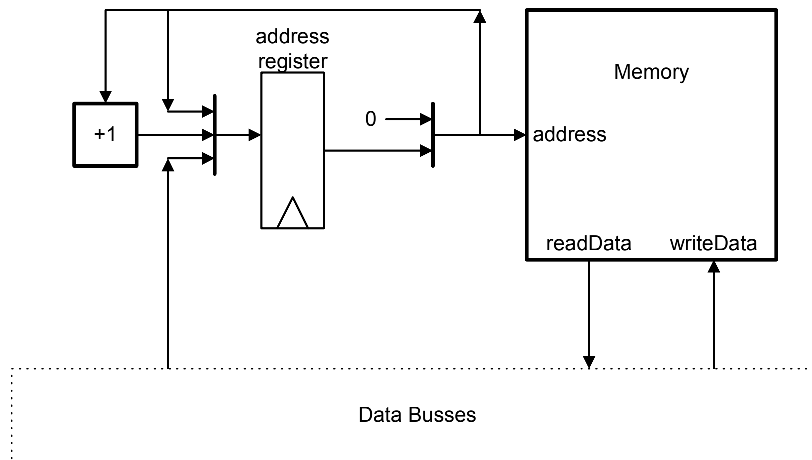

Figure 3 - Implementation of a RaPiD memory

Typically a RAM is loaded with data by first clearing the address register, then incrementing it each cycle that a write is performed. So, for example, data is typically written to location 0 on the first cycle of a loop and then to location 1 on the second cycle. The address can be cleared or loaded only at the beginning of a cycle before it is used and then incremented at the end of a cycle after it has been used. That is, operations on the address, if used, must be done in the order clear, use, increment.

The following program uses one RAM in each stage and loads the 0th location in each RAM with data from an external vector. That is, the 0th location of RAM[s] is loaded with the value X[s]. First the Pipe vector inData is loaded with the vector X starting with the last index. When the inData Pipe vector is full, it is written in parallel into

the RAMs at address 0. (For many algorithms, it is convenient to rename the elements of the vector so that element 0 is in stage **STAGE-1**. In this case, the input vector would be read starting with element 0.)

```
Pipe inData(1);
RAM data[STAGES];
For i;
for (i = 0; i < STAGES; i++) {
    Datapath {
        if (s==0) inData = X[STAGES-i-1];
        if (i.last) {
            data[s].address = 0;
            data[s] = inData;
        }
    }
}
```

The following program shows an alternative way to load data by reading it into a broadcast Pipe variable and writing it directly into the destination RAM, one value per iteration of the loop. Note that this reads the data into the same locations as the previous program. Also note the if condition **s==i**, which must be evaluated at run time. Fortunately this can be compiled very efficiently into a single control signal.

```
Pipe inData;
RAM data[STAGES];
For i;
for (i = 0; i < STAGES; i++) {
    Datapath {
        if (s==0) inData = X[i];
        if (s==i) {
            data[s].address = 0;
            data[s] = inData;
    }
}
```

The following program extends this example by loading an entire matrix into a set of RAMs, again one RAM per stage, and one column of the matrix per RAM. The external matrix is defined as **X[M][N]** where **N==STAGES** and **M < 64**, the number of locations in each RAM. Column j of X is loaded into the RAM in stage j. Each row is loaded using the first method of the previous example.

```
Pipe inData(1);
RAM data[STAGES];
For i, j;

for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        Datapath {
            if (i.first && j.first) data[s].address = 0;
            if (s==0) inData = X[i][STAGES-j-1];
            if (j.last) {
                data[s] = inData;
                data[s].address++;
            }
        }
    }
}

StreamIn inStream;
Pipe inData;
Word temp[STAGES];                  // needed for swap
RAM data[STAGES];
For i, j;

for (i = 0; i < N; i++) {
    for (j = 0; j < K; j++) {
        Datapath {
            if (j.first) data[s].address = 0;
            if (s==0) inData = inStream;
            temp = data[s];
            data[s] = inData;
            inData = temp;
            data[s].address++;
        }
    }
}
```

A read is performed at the start of every clock cycle, and a write is optionally performed at the end of the same cycle, to the same address. Memory can thus be used as a configurable-length shift register where the oldest data is read from the front of the shift register, and is then replaced by the newest data as it becomes the end of the shift register. The following program does this, where K is the length of the shift register in each stage, which is implemented using a RAM, and N*K is the total number of data values read. Here the address register is used as both the head and tail pointer. Of course this implementation starts out with an empty shift register and ends up with a full one, so a program that uses this shift register would have to deal with initialization and finalization.

Of course, if the data written back to the memory location depends on the data just read, then the cycle time will be very long. So, for example, if values are accumulated using memory, it must be done in a pipelined way where the value written back is written on the next cycle. If the pipelining/retiming algorithm included an understanding of the memories, then this could be done automatically. [4]

Threads

Typically RaPiD-C programs are composed of nested For loops that execute sequentially as we saw above. But there are times when it is convenient to have two independent For loops running at the same time, executing different but related computations in the datapath. This can be done in RaPiD-C by using multiple threads of execution, which is achieved by starting up two For loops running in parallel. The Datapath statements in the parallel For loops run concurrently. That is, the datapath is configured to execute the statements in the two For

[4] RAMs should be upgraded with a configurable Size register that would be used the increment the address modulo Size. If this were the case, then for this program we would configure the memory size to K, do away with the inner loop and execute the outer loop N*K times.

loops at the same time. Moreover, the statements in the two concurrent For loops can communicate via shared variables, in the same clock cycle. However, the program order in which these variables are written and read by parallel threads is important. That is, if one thread wants to read a variable written by another thread, then it should appear after that thread.

The Par construct is used to execute multiple parallel threads. The example above that copies one external vector to another can be written in a more straightforward way using multiple threads as follows:

```
Pipe InData(1);
For i, wait, j;
Par {
    thread:
        for (i = 0; i < N; i++) {
            Datapath {
                if (s==0) inData = X[i];
            }
        }
    thread:
        for (wait = 0; wait < STAGES-1; wait++) {
            Datapath { }                // Do nothing until inData is full
        }
        for (j = 0; j < N; j++) {
            Datapath {
                if (s==STAGES-1) Y[j] = inData;
            }
        }
}
```

In this program, the first thread performs all the input to the **`inData`** Pipe variable. The second thread first waits for **`STAGES-1`** cycles for the datapath to fill, and then outputs the N values to the output stream. Note that the first thread completes before the second thread. However, the overall Par statement does not complete until all the threads it has spawned have completed.

In this program, the second thread synchronized with the first by counting cycles until the **`inData`** Pipe was full. This can be cumbersome and often makes the program very complex. RaPiD-C provides Event variables which can be used to synchronize threads explicitly. A thread causes an Event to occur via the Signal statement and another thread can wait for a specified Event to occur via the Wait statement. Here is the same program written using Events:

```
Pipe InData(1);
For i, j;
Event Ready;
Par {
    thread:
        for (i = 0; i < N; i++) {
            if (i == STAGES-1) Signal(Ready);
            Datapath {
                if (s==0) inData = X[i];
            }
        }
    thread:
        Wait(Ready);
        for (j = 0; j < N; j++) {
            Datapath {
                if (s==STAGES-1) Y[i] = inData;
            }
        }
}
```

The semantics of Signal/Wait are simple: the first iteration after the Wait statement is executed on the same clock cycle as the iteration that executes the Signal. In this example, the Signal is executed when **`i==STAGES-1`**, which

means that the first iteration of the j For loop, i.e. when `j==0`, is executed when `i==STAGES-1`. This is the same timing as the previous program, but the synchronization here is more obvious. Moreover, the index computations are simpler and the external vectors are referenced only once each, so the compiler generates only one input stream and one output stream.

Events can be used to implement barriers as well. In this case, each thread issues a Signal followed by a Wait. When both threads reach their Signal statement, then both threads execute the iteration after the Wait simultaneously.

Boolean Variables

A second type of variable is the Boolean control variable, which are declared as type Bit. For example, the result of a comparison can be stored in a Bit variable and used later as the condition of an if statement. Status results from combinational units like ALUs can also be stored in Bit variables and used later as control in if statements. In the following example, the 32-bit result of a multiply is added into a 32-bit accumulator.

```
   Bit Carry;

   TempLo[s] = ramWeight[s] * pipeIn; // The low half
// The result of the above is a long word
   TempHi[s] = TempLo[s].hi;  // The high half
   ACLo[s] = ACLo[s] + TempLo[s];
   Carry = ACLo[s].cout;
   ACHi[s] = ACHi[s] + TempHi[s];
   If (Carry) ACHi[s] = ACHi[s] + 1;
```

This example uses the built-in ALU and Multipliers that the compiler provides. The .hi and .cout give access to the status values produced by these function units. Using generic versions of these components allows the programmer to explicitly refer to all control and status input/outputs of function units.

If a Boolean variable is used before it is assigned in a clock cycle, then a Boolean register is inferred. This way simple state machines can be generated in the control path. This is often used to reduce the number of control bits.

Bit variables are communicated from stage to stage using BitPipes, which are the control analog to data Pipes. An interesting use of Bit variables occurs in the extended matrix multiply program (included at the end of this document). A BitPipe is used as a global flag variable, which is used as a double-buffer index. The load thread uses this flag to read a matrix into the appropriate buffer, while the compute thread uses the flag to access the opposite buffer.

Always and DC statements

The compiler is very careful when generating control to make sure that operations are enabled only when the containing loop is being executed. For example, in the following code, all control signals that enable operations in this loop are predicated on the condition i.live, which makes sure that the operations execute only when the loop is active. In this fragment, taken from the example program doublePipe2, the always statement is used to indicate that the transfer of the Pipe variable to register to Pipe variable can happen on every cycle of the computation:

```
for (i = 0; i < NINPUTS; i++) {
   Datapath {
      always if (s<STAGES-1) {
         temp[s] = reg[s];
         reg[s] = data;
         data = temp[s];
      }
      X[s] = X[s] + reg[s];
   }
}
```

In many cases, this extra predicate is not needed since the operation can be executed on all cycles without affecting the overall computation. In this case, the **always** statement is used to indicate that the statements in the enclosed block are in effect for the entire program. This typically reduces the number of control bits.

The always directive can qualify any statement within the Datapath instruction. Although the Datapath instruction is executed only when this loop executes, the statements within the always block are executed for all time and thus require no control signals. By contrast, the assignment to X[s] in this example is controlled by a control signal that is asserted only when the i loop is being executed. Judicious use of the always statement can reduce the amount of control considerable.

If statements can be used in the always block to specify exactly when the operations are to be executed. For example, in the following code:

```
for (i = 0; i < NINPUTS; i++) {
   Datapath {
      if (i.first) temp1[s] = reg[s];
      else         temp2[s] = reg[s];
   }
}
```

the then clause is controlled by `(i.live && i.first)` and the else clause is controlled by `(i.live && !i.first)`. That is, two control signals are required to execute this code. Enclosing the if statement in an always block removes the `i.live` predicate, leaving only the `i.first` predicate. This means the then clause is controlled by `i.first` and the else clause is controlled by `!i.first`. Note that this means the else clause is executed even when the program is not in the i loop.

Note that the following program, the control signals are specified precisely by the programmer within the always block so that the assignment to temp happens on every cycle of the program, while the assignment to temp2 only happens in the i loop.

```
for (i = 0; i < NINPUTS; i++) {
   Datapath {
      always {
         temp[s] = reg[s];
         if (i.live) temp2[s] = reg[s];
      }
   }
}
```

Streams should not be read or written for all cycles using an always block. The compiler inserts an initialization loop at the beginning of every program that executes STAGES number of cycles. If a stream is read using an always block, or equivalently in a vthread, then it is read during this initialization loop and the values thrown away. Thus, even though the stream addresses are defined in a vthread, the actual stream read should be performed in a regular thread.

The `dc( )` statement gives information to the compiler about conditions that cannot happen. The compiler can figure out some of these don't care conditions itself. For example, if a for i loop and a for j loop are executed sequentially, then the compiler knows that `(i.live && j.live)` is a don't care condition, that is, a condition that can never be true. However, the compiler cannot deduce the don't care conditions for loops that are in different

threads. Thus, if the programmer knows that the i loop in one thread never overlaps the execution of the j loop in another thread, the statement:

```
dc(i.live && j.live)
```

can be placed at the beginning of the program, after the declarations and before the program starts. More generally, the programmer can tell the compiler about any condition that cannot happen, or which the compiler can ignore. Without this information, the compiler must ensure that the control signals are generated correctly for situations that never happen, which can cause an increase in the number of control signals and the program size.

Using the dc statement typically requires a fairly good knowledge of how the compiler generates the control bits. Otherwise, the programmer can try to identify as many don't care conditions as possible and hope for the best.

BackPipes

Pipe variables allow communication from one stage to the next, but do not allow communication in the reverse direction. BackPipes are used to do this and have the same semantics as Pipes, except that the initialization, and thus the data flow, takes place in the reverse direction. Using BackPipes can cause circular data dependencies, which is necessary in some cases, for example, the IIR filter. If such a circular dependency contains 0 delay, for example, if a 0-delay Pipe and BackPipe are used, then this is a combinational loop which cannot be compiled. If there is only one Pipe delay in the loop, then the clock cycle will be constrained to the combinational delay around this loop, which may be very slow. In such cases, the programmer can increase the performance of the datapath by inserting more Pipe delays in the loop or reducing the combinational delay in the loop. Following is an example of using a BackPipe variables. In this program, data is sent down the datapath via the **right1** Pipe, then back to the beginning using the **left** BackPipe, and finally back to the end using the **right2** Pipe. Since the Pipes are all delayed Pipes, they are vectors that shift one stage per cycle. Note that if the Pipes are all made into simple Pipes, then there would be zero delay through the datapath. Since there is no circular data dependency, the compiler would be able to pipeline this path correctly. Note also that this works for simple BackPipes in spite of the **for (s=0; s<STAGES; s++)** semantics of the Datapath statement.

```
#define DELAY (3*(STAGES-1))   // The delay through the 3 pipes
Pipe right1(1), right2(1);   // Pipe going right
BackPipe left(1);            // Pipe going left
For i;

for (i=0; i<XLength+DELAY; i++) {
    Datapath {
        // Get input and put into pipe going right
        if (s==0 && i < XLength) right1 = inX[i];
        // At right end of datapath, turn data around and send left
        if (s==STAGES-1) left = right1;

        // At left end, turn around again and send right
        if (s==0) right2 = left;
        // Finally, output the results
        if (s==STAGES-1 && i >= DELAY)
            outY[i-DELAY] = right2;
    }
}
```

Generic Function Units

The Rapid-C compiler compiles references to C operators like +, -, *, < and << into function unit references. The compiler assumes that these function units are available in the datapath. Although this is convenient, adding new function units or changing the operation of function units requires substantial compiler modifications. The Rapid-C compiler now support generic function units, which are classified as combinational (CombUnit) or sequential (StUnit). The function unit is defined using Verilog, which is used to simulate/synthesize the function unit in the datapath. The function unit can then be declared and used in the Rapid-C program with an interface that matches this definition. We will describe how to use generic function units in Rapid-C programs here. Detailed instructions

on how to create the Verilog models and include them in the compiler are described in the RPD#2001-007 document (http://www.cs.washington.edu/research/lis/rapid/internal/doc/2001/RPD2001-007/rpd2001-007.pdf)

The generic unit inputs and outputs are divided into four categories: Data outputs, Control outputs (status bits), Data inputs and Control inputs. The generic units are declared by giving the name of the unit along with these four lists.

```
CombUnit(<name>, (<Data outputs>), (<Control outputs>),
                 (<Data inputs>), (<Control inputs>));

StUnit(<name>,   (<Data outputs>), (<Control outputs>),
                 (<Data inputs>), (<Control inputs>));
```

If there are no signals in a category, then the corresponding parentheses are left empty.

Combinational Units

Combinational and function units differ in how they are used. Combinational units are invoked as part of the code and each invocation uses a different instance of the corresponding function unit. State units by contrast are instantiated at the beginning of the program and their input and output signals assigned within the program. Combinational units are called using the synax:

```
((<Data outputs>), (<Control outputs>)) = <name>((<Data inputs>), (<Control inputs>))
```

Here is an example of multiply-accumulate combinational unit. This unit has three data inputs and one output and performs the computation **R = M1*M2 + A**. This function unit has no control signals, but a carry in/out could easily be added to the control inputs/outputs.

```
CombUnit(mac, (R), (), (M1,M2,A), ());

Word result;

   if (i==0) result = 0;
   ((result), ()) = mac((inData, Tap[s], result), ());
```

Sequential Units

Sequential units are declared like combinational units, but a used very differently. Each state unit used in a program is declared at the beginning of the program, just like RAMs which are special state units. The inputs and outputs of the state unit are specified within the program on a cycle by cycle basis. If no assignment is made to a data or control input, then the signal is given the default value, if specified, or a don't care value. Default values for control inputs must be given using an = expression. For example, Enable=0 indicates that the control input Enable is by default off unless explicitly given. If an input is always assigned the same value, the **always** statement can be used. The following is an example of a time-multiplexed FIR program that uses an enabled configurable-length shift register.

```
// Configurable-length shift register
// Size = length of shift register
// Enable = assert to shift
// Out = current output value
// In = value shifted in if Enable=1
StUnit(clsr, (Out), (), (Size, In), (Enable=0));

void
fir(Word arrTaps[NW], Word arrData[NX], Word arrResults[NX-NW+1])
{
  clsr Inputs[STAGES];
  clsr Taps[STAGES];

   Pipe pipeTaps(1), pipeData(0);
   Pipe pipeResults(0);

   Word AC[STAGES];
   Word TapsSize[STAGES], InputSize[STAGES], Forward[STAGES];

   For w, tm1, calc, tm2;

   for (tm1=0; tm1 < W_PER_STAGE; tm1++) {
     for (w=0; w < NW; w+=W_PER_STAGE) {
       Datapath {
        always { // clsr size is always the same
          Inputs[s].Size = W_PER_STAGE-1;
          Taps[s].Size = W_PER_STAGE;
        }
        if (s==0) pipeTaps = arrTaps[w+tm1];
        Taps[s].In = pipeTaps;
        if (w.last) {
          Taps[s].Enable = 1;
        }
       }
     }
   }

   for (calc=0; calc < NX + STAGES; calc++) {
     for (tm2=0; tm2 < W_PER_STAGE; tm2++) {
       Datapath {
        Inputs[s].Enable = 1;
        Taps[s].Enable = 1;
        Taps[s].In = Taps[s].Out;
        if (tm2.first) {
          if (s==0) pipeData = arrData[calc];
          Inputs[s].In = Forward[s];
          Forward[s] = pipeData;
          AC[s] = 0;
        } else {
          Inputs[s].In = Inputs[s].Out;
        }

        pipeData = Inputs[s].Out;

        AC[s] += Inputs[s].Out * Taps[s].Out;

        if (tm2.last) {
          if (s==0) pipeResults = AC[s];
          else     pipeResults = pipeResults + AC[s];
          if (s==STAGES-1) arrResults[calc] = pipeResults;
        }
       }
     }
```

Currently Available Function Units

To be completed

Compiling and Simulating RaPiD-C Programs

The RaPiD-C compiler takes a RaPiD-C program (extension **`.rc`**) and produces a Verilog simulation file that can be executed by the Verilog simulator. The interface to external memory is provided via a C++ driver program, which first initializes memory with the appropriate data, then calls the Verilog simulator to execute the compiled RaPiD-C program, and then finally analyzes the results that have been stored in memory by the computation to determine if they are correct. As a side-effect, the Verilog simulation produces a debugging output file which gives the values of all the RaPiD-C variables on every clock cycle of the simulation. The programmer can look at this execution trace using an Emacs window to step through the computation one clock cycle at a time.

The compiler, simulator and all related files can be found in the **`/projects/lis/rapid/rcc/`** directory on the research Suns. The following example application can be found in the **`rcc/apps/examples/simplepipe/`** directory. To try it out, simply copy it to your directory and use the **`gmake`** to compile and simulate it. The compilation and simulation is done by a Makefile which understands where all the necessary files are in the rrc directory. Let's start with the simplest RaPiD-C program, called **`simplepipe.rc`**, which simply copies data from one vector in external memory to another. (This and other programs found in this manual can be found in the **`rcc/apps/examples`** directory.)

```
//****************************************************************************
// Example program showing how to implement an simple broadcast to copy  // data from input to output
//
//****************************************************************************
#define STAGES 4                   // Define the number of stages #include "rapidb.h"
                                           // Always include
#define XLength 20                 // Length of input vector X #define YLength Xlength
                                           // Length of output vector Y
void simplePipe(Word inX[XLength], Word outY[YLength]) {
     InPipe inData;                     // Broadcast Pipe
     For i;

     for (i=0; i<XLength; i++) {
          Datapath {
               if (s==0) inData = inX[i]; // Read data at the beginning
               if (s==STAGES-1) outY[i] = inData; // Write at the end
          }
     }
}
```

We have wrapped the RaPiD-C program in a function definition. The C++ driver program executes the simulation of the RaPiD-C program by executing this function. The parameters to this function are the external memory arrays that are read or written by the RaPiD-C program. In this example, we provide an input array and one output array. These arrays can be multi-dimensional and are referenced directly in the RaPiD-C program. As shown, the size of the arrays must be specified and must match the declaration of this function given in the C++ driver program.

```
#include <assert.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#define STAGES 4
#include "rapidb.h"
#define XLength 20                // Length of input vector X
#define YLength XLength                // Length of output vector Y

void simplePipe(Word *, Word *);
int main()  {
     int i;
     Word inX[XLength], outY[YLength];

     srandom(1);
     // Randomly fill input array
     for (i=0; i<XLength; i++)
          inX[i] = RAND(100) + 1;

     // Call the RaPiD-C program
     simplePipe(inX, outY);

     // Print results
     printf("    inX =");
     for (i=0; i<XLength; i++)
          printf(" %4d", inX[i]);
     printf("\n");
     printf("    outY =");
     for (i=0; i<YLength; i++)
          printf(" %4d", outY[i]);
     printf("\n");
   // Verify results
     BOOL ok = TRUE;
     for (i=0; i<XLength; i++)
          ok &= (inX[i] == outY[i]);
     if (!ok)
          fprintf(stderr, "Verification failed!\n");
     else
          fprintf(stderr, "Verification succeeded.\n");
}
```

The C++ driver program, shown above, is an arbitrary C++ program which at some point calls the RaPiD-C function to execute the Verilog simulation. Note that #define's and #include's used by both programs are normally put in a .h file.

To compile and execute this program, the following GNU Makefile is used which includes the **`Makefile.app`** which understands that RaPiD-C programs end with **`.rc`**.

```
RAPID_ROOT=/projects/lis/rapid/rcc
simplePipe: simplePipetest.o simplePipe.o
     gcc -o simplePipe simplePipetest.o simplePipe.o

include $(RAPID_ROOT)/apps/Makefile.app
```

Compiling the program produces the executable program **`simplePipe`**, which when executed, produces the following output:

```
Executing verilog.  Please be patient.  This may take a while. (warnings/errors are being written to simplePipe.vout)
     inX =   97    7   48   91   36   94   66    3   52   21   94   21   38   80   29   27   36   13   71   74
     outY =   97    7   48   91   36   94   66    3   52   21   94   21   38   80   29   27   36   13   71   74
Verification succeeded.
```

The compilation and simulation process produces a number of intermediate and debugging files which are stored in a subdirectory named **`RAPID`**. These include a **`.cc`** file which is the C++ wrapper program produced from the **`.rc`** program file, several **`.s`** and **`.bin`** files which are the assembly programs and executables for the context and address generators, a **`.v`** file which contains the Verilog file that is simulated, and a **`.vout`** file which is the log of the Verilog simulation. The most important file, the **`.sim`** file is put in the same directory as the source files. This file contains the debugging output produced by the simulation and contains the values of all the variables for all the simulation cycles. The value of each variable is shown *at the end of the cycle just before the clock ticks*. This means that we do not see the new value of registers until the following cycle.

When the simulation debug file is opened using Emacs, the window is resized so that each page down command advances to the next clock cycle. The window may have to be widened and the font reduced to get everything in a window. The window may have to be resized manually as well.

Part of the simulation debug output file for the simplePipe program is shown below. A number of cycles are required for initialization so we begin with the first cycle when something ``happens''.

```
***Cycle #12
         0:     97,      97,      97,         (inData_00_00) --> (inData_02_00)
         0:     97,        ,        ,         (inX_i) --> (inX_i)
         0:      1,        ,        ,         (inX_i.Read_out) --> (inX_i.Read_out)
         0:       ,        ,        ,      97 (outY_20_i) --> (outY_20_i)
         0:       ,        ,        ,       1 (outY_20_i.Write_out) --> (outY_20_i.Write_out)
 ***Cycle #13
         0:      7,       7,       7,         (inData_00_00) --> (inData_02_00)
         0:      7,        ,        ,         (inX_i) --> (inX_i)
         0:      1,        ,        ,         (inX_i.Read_out) --> (inX_i.Read_out)
         0:       ,        ,        ,       7 (outY_20_i) --> (outY_20_i)
         0:       ,        ,        ,       1 (outY_20_i.Write_out) --> (outY_20_i.Write_out)
 ***Cycle #14
         0:     48,      48,      48,         (inData_00_00) --> (inData_02_00)
         0:     48,        ,        ,         (inX_i) --> (inX_i)
         0:      1,        ,        ,         (inX_i.Read_out) --> (inX_i.Read_out)
         0:       ,        ,        ,      48 (outY_20_i) --> (outY_20_i)
         0:       ,        ,        ,       1 (outY_20_i.Write_out) --> (outY_20_i.Write_out)
......................................................
***Cycle #31
         0:     74,      74,      74,         (inData_00_00) --> (inData_02_00)
         0:     74,        ,        ,         (inX_i) --> (inX_i)
         0:      1,        ,        ,         (inX_i.Read_out) --> (inX_i.Read_out)
         0:       ,        ,        ,      74 (outY_20_i) --> (outY_20_i)
         0:       ,        ,        ,       1 (outY_20_i.Write_out) --> (outY_20_i.Write_out)
```

In this program, there is the **`inData`** Pipe variable, and an input stream and an output stream generated by the compiler called inX and outY. For streams, both the data and the read or write control signal is shown. Many variables occur one per stage, for example arrays and Pipes, and so space is left to show the variable value in each stage. A z value indicates that the variable did not actually occur in the stage and it is useful to replace all the z's with spaces, as has been done here. Note that the input stream values occur only in the first stage and the output stream values occur only in the last stage.

Since inData is a Pipe variable, it shows up in each stage except the last. The value shown is the value last assigned to the Pipe variable in that stage, which becomes the initial value of the variable in the next stage. (For some reason, the Pipe variable value is not shown in the last stage, probably because it is not forwarded to the next stage and thus does not exist.)

Input streams provide a new data value in response to the stream **`Read`** signal and the new value shows up on the same cycle. Output stream data values are written to the output stream on the same cycle that the stream **`Write`** signal is asserted. In this simulation, the input Read signal is asserted on Cycle 12 to access the first data value. This data is assigned to the broadcast Pipe variable **`inData`** which propagates it unchanged to the last stage, where it is assigned to the output stream. This continues one new data value per cycle until the last cycle, Cycle 31.

This simulation uses the RaPiD-C broadcast communication model so that the programmer can easily debug the program. The actual implementation is pipelined by a later phase of the compiler so that it may in fact take data many cycles to get through the datapath. Since this pipelining and retiming is done automatically and can be proven correct, this form of the simulation is all that the programmer really needs to do.

Program Examples

doublePipe

The following program uses a Pipe variable (vector) **inData** to copy data from the external vector inX to the external vector outY. This shows the use of parallel threads to synchronize the reading and writing of data.

```
#include "doublePipe.h"

void doublePipe(Word inX[XLength], Word outY[YLength]) {
Pipe inData(1);                // shift register
For inputI, outputWait, outputI;
Par {
    // This thread just reads data into the beginning of the data pipe
    thread:
        for (inputI=0; inputI<XLength; inputI++) {
            Datapath {
                if (s==0) inData = inX[inputI];
            }
        }
    thread:
        // Wait for shift register to fill
        for (outputWait=0; outputWait<STAGES-1; outputWait++) {
            Datapath { }
        }
        // Now output all the inData
        for (outputI=0; outputI<YLength; outputI++) {
            Datapath {
                if (s==STAGES-1) outY[outputI] = inData;
            }
        }
    }
}
```

The simulation trace for this program follows on the next page. There are two values shown for the delayed Pipe variable. The first (with suffix `00`) is the value last assigned to the Pipe variable in this clock cycle. The second (with suffix `01`) is the value saved from the previous cycle (the contents of the register) which is used as the initial value in the next stage. The `01` value of one stage is always the same as the `00` value of the next stage. In this example, we can see the input value shifting through the **inData** vector one stage per cycle until is reaches the last stage where it is written to the output stream. Note that the output Write signal is asserted at exactly the right time. On the last cycle, the Pipe variable has the last input data in each stage. This is because the input stream has stopped being read and the last value in the stream is still asserted on the input.

```
***Cycle #12
         0:     97,      x,      x,         (data_00_00) --> (data_02_00)
         0:      x,      x,      x,         (data_00_01) --> (data_02_01)
         0:     97,       ,       ,         (inX_inputI) --> (inX_inputI)
         0:      1,       ,       ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,       ,       ,       x (outY_20_outputI) --> (outY_20_outputI)
         0:       ,       ,       ,       0 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
 ***Cycle #13
         0:      7,     97,      x,         (data_00_00) --> (data_02_00)
         0:     97,      x,      x,         (data_00_01) --> (data_02_01)
         0:      7,       ,       ,         (inX_inputI) --> (inX_inputI)
         0:      1,       ,       ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,       ,       ,       x (outY_20_outputI) --> (outY_20_outputI)
         0:       ,       ,       ,       0 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
 ***Cycle #14
         0:     48,      7,     97,         (data_00_00) --> (data_02_00)
         0:      7,     97,      x,         (data_00_01) --> (data_02_01)
         0:     48,       ,       ,         (inX_inputI) --> (inX_inputI)
         0:      1,       ,       ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,       ,       ,       x (outY_20_outputI) --> (outY_20_outputI)
         0:       ,       ,       ,       0 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
 ***Cycle #15
         0:     91,     48,      7,         (data_00_00) --> (data_02_00)
         0:     48,      7,     97,         (data_00_01) --> (data_02_01)
         0:     91,       ,       ,         (inX_inputI) --> (inX_inputI)
         0:      1,       ,       ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,       ,       ,      97 (outY_20_outputI) --> (outY_20_outputI)
         0:       ,       ,       ,       1 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
 ***Cycle #16
         0:     36,     91,     48,         (data_00_00) --> (data_02_00)
         0:     91,     48,      7,         (data_00_01) --> (data_02_01)
         0:     36,       ,       ,         (inX_inputI) --> (inX_inputI)
         0:      1,       ,       ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,       ,       ,       7 (outY_20_outputI) --> (outY_20_outputI)
         0:       ,       ,       ,       1 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
..............................................
***Cycle #34
         0:     74,     74,     74,         (data_00_00) --> (data_02_00)
         0:     74,     74,     74,         (data_00_01) --> (data_02_01)
         0:     74,       ,       ,         (inX_inputI) --> (inX_inputI)
         0:      0,       ,       ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,       ,       ,      74 (outY_20_outputI) --> (outY_20_outputI)
         0:       ,       ,       ,       1 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
```

doublePipe2

The following program does the same thing as the previous program, that is, it copies one external vector to another using a delayed Pipe variable. In this case, however, the delayed Pipe variable is implemented using a simple Pipe variable and an explicit register called **`reg[s]`**. **`temp[s]`** is a temporary variable that is used in each stage to swap the Pipe value with the value stored in **`reg`**.

This program points out a couple things. First, data is communicated between the parallel threads in the same clock cycle. The first thread assigns a value to the **`data`** Pipe variable, which is then used in the second and third threads. Second, if we want an operation to happen on every cycle of the computation, or if we don't care if it happens on every cycle of the computation, we can use the **`always`** statement to indicate this. This can be used to modify any Datapath statement so that it is executed every cycle. This can reduce the control complexity since it avoids all dynamic control. It can also make it easier to write a program. If the programmer wanted to conditionally shift the data vector, then there would be an if statement instead of an always statement.

```
void doublePipe2(Word inX[XLength], Word outY[YLength]) {
Pipe data;                        // Broadcast pipe to communicate stage to stage
Word reg[STAGES];                // One register per stage for pipeline
Word temp[STAGES];                // Temporary value for swap - not a register

For inputI, outputWait, outputI;
For allI;
Par {
     thread:
          for (inputI=0; inputI<XLength; inputI++) {
               Datapath {
                    if (s==0) data = inX[inputI];
               }
          }
     thread:
          Datapath {
               always if (s<STAGES-1) {
                    temp[s] = reg[s];
                    reg[s] = data;
                    data = temp[s];
               }
          }
     thread:
          {
               // Wait for shift register to fill
               for (outputWait=0; outputWait<STAGES-1; outputWait++) {
                    Datapath {  }
          }
               // Now output all the data
               for (outputI=0; outputI<YLength; outputI++) {
                    Datapath {
                         if (s==STAGES-1) outY[outputI] = data;
                    }
               }
          }
     }
}
```

In the following simulation debug output, it can be seen that the values shown are the values of variable just before the clock tick at the end of a cycle. In Cycle 12, the first value is available from the input stream. This value is written to **`reg[0]`**, but this happens only when the clock tick at the end of the cycle occurs, so we do not see the value until Cycle 13. The value of **`reg[0]`** is put on the **`data`** Pipe variable and then updated with the new stream input value.

```
***Cycle #12
         0:      x,       x,       x,         (data_00_00) --> (data_02_00)
         0:     97,        ,        ,         (inX_inputI) --> (inX_inputI)
         0:      1,        ,        ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,        ,        ,       x (outY_20_outputI) --> (outY_20_outputI)
         0:       ,        ,        ,       0 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
         0:      x,       x,       x,         (reg_00) --> (reg_02)
 ***Cycle #13
         0:     97,       x,       x,         (data_00_00) --> (data_02_00)
         0:      7,        ,        ,         (inX_inputI) --> (inX_inputI)
         0:      1,        ,        ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,        ,        ,       x (outY_20_outputI) --> (outY_20_outputI)
         0:       ,        ,        ,       0 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
         0:     97,       x,       x,         (reg_00) --> (reg_02)
 ***Cycle #14
         0:      7,      97,       x,         (data_00_00) --> (data_02_00)
         0:     48,        ,        ,         (inX_inputI) --> (inX_inputI)
         0:      1,        ,        ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,        ,        ,       x (outY_20_outputI) --> (outY_20_outputI)
         0:       ,        ,        ,       0 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
         0:      7,      97,       x,         (reg_00) --> (reg_02)
 ***Cycle #15
         0:     48,       7,      97,         (data_00_00) --> (data_02_00)
         0:     91,        ,        ,         (inX_inputI) --> (inX_inputI)
         0:      1,        ,        ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,        ,        ,      97 (outY_20_outputI) --> (outY_20_outputI)
         0:       ,        ,        ,       1 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
         0:     48,       7,      97,         (reg_00) --> (reg_02)
 ***Cycle #16
         0:     91,      48,       7,         (data_00_00) --> (data_02_00)
         0:     36,        ,        ,         (inX_inputI) --> (inX_inputI)
         0:      1,        ,        ,         (inX_inputI.Read_out) --> (inX_inputI.Read_out)
         0:       ,        ,        ,       7 (outY_20_outputI) --> (outY_20_outputI)
         0:       ,        ,        ,       1 (outY_20_outputI.Write_out) --> (outY_20_outputI.Write_out)
         0:     91,      48,       7,         (reg_00) --> (reg_02)
```

Simple FIR Filter

The following program executes a FIR filter with 16 taps. This program follows the method presented before. First the weights are loaded into the **wTaps** array. The two parallel threads are started, one to input data into the **inpipeData** Pipe vector, and one to compute the results and store them in the output array. A Signal/Wait is used to synchronize these two threads.

```
#define STAGES 16
#include "rapidb.h"
#define NW STAGES
#define NX 70

void firsm(Word arrTaps[NW], Word arrData[NX], Word arrResults[NX-NW+1]) {
Pipe inpipeTaps(1), inpipeData(1);
Pipe outpipeResults;

Word wTaps[NW];

For w, fill, out;
Event eventOut;

// Load the weights in wTaps
for (w = 0; w < NW; w++)
     Datapath {
          if (s==0) inpipeTaps = arrTaps[w];
          if (w.last) wTaps[s] = inpipeTaps;
     }

Par {
     thread:
          // Fill the input data pipe
          for (fill = 0; fill < NX; fill++)
               Datapath {
                    if (s==0) inpipeData = arrData[fill];
                    if (fill==NW-1) Signal(eventOut);
               }
     thread:
          // Do the computation
          Wait(eventOut);
          For (out = 0; out < NX-NW+1; out++)
               Datapath {
                    if (s==0) outpipeResults = 0;
                    outpipeResults += inpipeData * wTaps[s];
                    if (s==NW-1) arrResults[out] = outpipeResults;
               }
          }
     }
```

Simple Matix Multiply

This program does a matrix multiply of A x B where the matrix B can fit into the RAMs, one column per stage. The B matrix is first loaded into the BRam and then the result matrix is computed one row at a time. That is, the first row of A is multiplied by the matrix B to form the first row of C and so forth.

```
#define STAGES 4
#include "rapidb.h"

// Note: M cannot exceed 32 since RAMs hold max 32
// values and will roll over (mod addressing) if > 32
// Note: M + 2 >= N, since the alg is not set up
// to handle the case where the number of results
// exceeds the time to compute the next result
#define L 4
#define M 4
#define N STAGES

// This version of matrix multiply assumes that the B-matrix fits in
// the RaPiD array.
void  limited(Word A[L][M], Word B[M][N], Word C[L][N]) {
Pipe pipeIn;              // Simple pipe for all inputs
Pipe pipeOut(1);          // Results are shifted out

Ram BRam[N];              // Stores the B matrix
Word result[N];

For Wi, Wj;               // Indices for loading weights
For Ai;               // Iterates over the rows in A (and rows of C)
For Aj;               // Iterates over the columns of A
For Ci, Cj;               // Indices for sending output to C

Event OutputReady;        // Signals when an output row is ready

Par {
     thread:      // Preload the B-matrix into the RAMs
          for (Wi = 0; Wi < M; Wi++) {
               for (Wj = 0; Wj < N; Wj++) {
                    Datapath {
                         if (Wi.first && Wj.first) BRam[s].address = 0;
                         if (s==0) pipeIn = B[Wi][Wj];
                         if (Wj==s) BRam[s] = pipeIn;       // Store value in RAM
                         if (Wj.last) BRam[s].address++; // Go to the next row in RAM
                    }
               }
          }
          // Compute the output C matrix one row at a time
          for (Ai=0; Ai < L; Ai++) {
               // Vector multiply a row with a column
               // Note that (address==j)
               for (Aj=0; Aj < M; Aj++) {
                    Datapath {
                         if (Aj.first) {
                              BRam[s].address = 0;
                              result[s] = 0; // Initialize results
                         }
                         if (s==0) pipeIn = A[Ai][Aj];
                         result[s] += BRam[s]*pipeIn;
                         // On the last iteration of the row, latch the output
                         // and tell the output thread to proceed
                         if (Aj.last) {
                              Signal(OutputReady);
                              pipeOut = result[s];
                         }
                         BRam[s].address++;
                    }
               }
          }
     thread:
          // Output results.  Each signal tells this process that the next
          // row is available in the output Pipe vector
          for (Ci=0; Ci < L; Ci++) {
               Wait (OutputReady);
               for (Cj=N-1; Cj >= 0; Cj--)
                    // Row comes out in reverse order
                    Datapath {
                         if (s==N-1) C[Ci][Cj] = pipeOut;
                    }
          }
     }
}
```

General Matix Multiply

In the general version of matrix multiply, the result matrix is computed as a series of tiles. Each tile of the result matrix computed as the sum of several matrix multiplies of tiles in the A and B matrices. While each component matrix multiply is performed, the next tile of B that will be need to perform the next component matrix multiply is loaded. Two RAMs are used and swapped back and forth in a double-buffered way.

This program shows the use of a Boolean variable to keep track of the double-buffering of the B matrix. This variable tells each thread which RAM to use. At the end of each tile, this variable is switched so that each thread references a different RAM.

```
#define STAGES 4
// Matrices are A[L][M] x B[M][N] = C[L][N]
#define L 8
#define M 8
#define N 8
// Tiles are A[Q][R] x B[R][S] = C[Q][S]
#define Q 4
#define R 4

#define S STAGES
#include "matmult.h"
#include "rapidb.h"

void  matmult(Word A[L][M], Word B[M][N], Word C[L][N]) {
     InPipe inpipeA, inpipeB;
     OutPipe outpipeC(1);
     Ram ramB[2][S], ramC[S];

     Word wAccum[S], wSum[S];
     Word wMult[S], wMultiplier[S];

     BitPipe bParity;

     For init;
     For f,g,h,i,j,k;
     For f1,g1,h1,i1,j1,k1;
     For f2,g2,h2,i2,j2,k2;
     For f3, g3;

     Event eventOut;
     Event eventComp, eventPreload;

     //   Word SIM_CONTROL;    Word SIM_RAMEND = R-1;

     // Initialize the parity control bit.  Thereafter, the parity bit
     // is toggled when we start a new tile
     Datapath {
          if (s==0) bParity =  1;
          always if (s==0 && i1.first && j1.first) bParity = !bParity;
     }
```

```
    Par {
        thread:
            // This loop loads the B matrix tiles into the RAMs just before they
            // are used in the computation using double-buffering
            for (f=0; f < L; f+=Q) { // Repeat for each C row
                for (g=0; g < N; g+=S) { // Tiling across -> one C row
                    for (h=0; h < M; h+=R) { // Tiling down -> one C tile
                        // Read one tile RxS in size
                        for (j=h; j < h+R; j++) {
                            for (k=g; k < g + S; k++) {
                                Datapath {
                                    if (s==0) inpipeB = B[j][k];
                                    if (j.first && k.first) {
                                    if (!bParity) ramB[1][s].address = 0;
                                    else          ramB[0][s].address = 0;
                                    }
                                    if (k==g+s) {
                                        if (!bParity) ramB[1][s] = inpipeB;
                                        else          ramB[0][s] = inpipeB;
                                    }
                                    if (k.last && ! j1.live) {
                                        if (!bParity) ramB[1][s].address++;
                                        else          ramB[0][s].address++;
                                        }
                                } // Datapath
                            }
                        }
                        Signal(eventComp);  // Barrier at end of tile
                        Wait(eventPreload);
                    }
                }
            }
        thread:
            // This loop does the computation, generating tiles in C
            for (f1=0; f1 < L; f1=f1+Q) {
                for (g1=0; g1 < N; g1=g1+S) {
                    for (h1=0; h1 < M; h1=h1+R) {
                        // One C tile
                        Signal(eventPreload); // Barrier at beginning of tile
                        Wait(eventComp);
                        // Process one A tile
                        for (i1=f1; i1 < f1+Q; i1++) {
                            for (j1=h1; j1 < h1+R; j1++) {
                                Datapath {
                                    if (s==0) inpipeA = A[i1][j1];
                                    if (j1.first) {
                                        if (!bParity) ramB[0][s].address = 0; // Use opposite parity
                                        else          ramB[1][s].address = 0; // from loop above
                                    }
                                    if (!bParity) wMultiplier[s] = ramB[0][s];
                                    else          wMultiplier[s] = ramB[1][s];
                                    if (!bParity) ramB[0][s].address++;
                                    else          ramB[1][s].address++;
                                    if (k.last) {
                                        if (!bParity) ramB[1][s].address++;
                                        else          ramB[0][s].address++;
                                    }

                                    // We accumulate in a variable (wAccum[s]) and not memory because we
                                    // don't have enough time to read, add and write back
                                    // in one cycle
                                    if (j1.first) {
                                        wAccum[s] = 0;
                                    }
                                    wAccum[s] = wAccum[s] + wMultiplier[s] * inpipeA;

                                    // At the beginning of a tile, reset the output tile address
                                    if (i1.first && j1.first) ramC[s].address = 0;
                                    // Done accumulating a row: add to C matrix in ramC
                                    // Here we can add to memory because pipelining will save us??
                                    if (i1.last && j1.last) {
                                        if (h1.first)      ramC[s] = wAccum[s];
                                        else if (!h1.last) ramC[s] = ramC[s] + wAccum[s];
                                    // On the last iteration, we are done and can send the result
                                    // to the output
                                        else if (h1.last) {
                                                  Signal(eventOut);
                                                  outpipeC = wAccum[s] + ramC[s];
                                        }
                                    ramC[s].address++;                  }
                                } // Datapath

                            }
                        }
                    }
                }
            }

        thread:
            // Output loop
            for (f2=0; f2 < L; f2=f2+Q) {
                for (g2=0; g2 < N; g2=g2+S) {
                    for (i2=f2; i2 < f2+Q; i2++) {
                        Wait(eventOut);
                        for (k2=g2+S-1; k2 >= g2; k2--) {
                            // Result shift out in reverse order
                            Datapath {
                                if (s==S-1) C[i2][k2] = outpipeC;
                            }
                        }
                    }
                }
            }
    }
}
```

Debugging RaPiD-C Programs that Won't Compile

The Rapid-C compiler does an amazing job compiling complex Rapid-C programs. However, it was never meant to be an industrial strength compiler and thus it does not always give error messages that tell what happened and sometimes even crashes with no warning. Trying fix programs when this happens can be frustrating. Getting help from an experienced Rapid-C programmer is probably best, but here are things you can do to try to figure what is going on.

1. Reduce the program to determine what is causing the compiler to fail. Use comments to isolate the line that is causing the problem.
2. The RAPID subdirectory contains a file called <yourprogram>.debug The compiler writes all sorts of information to this file as it compiles your program. You can often tell what happened by what is in this file.
3. For the really brave, you can use gdb to debug the compiler. To do this, copy the rbc executable you used (typically in the rcc/bin directory specified by your Makefile) to your RAPID subdirectory. Go to the RAPID directory and start gdb on this rbc file. (Emacs gdb mode is a nice way to execute gdb.) Go to the line in the shell command window where the rbc program was executed by the makefile. Cut and paste this into gdb, substituting "run" for "rbc". This will execute the compiler, and will allow you to check the stack after it fails, and rerun after setting breakpoints.