THE μPD7281 PROCESSOR

BY TOM JEFFERY

A non–von Neumann chip designed for high-speed parallel processing of images

THE TRADITIONAL COMPUTER, as formalized by John von Neumann, has a central processing unit (CPU) that accesses instructions residing in the memory with the data. But fetching an instruction from memory, decoding the instruction, fetching data from memory, and storing results, over and over, instruction after instruction, pixel by pixel, slows a system down. For image processing, the fact that similar operations are repetitively performed on all the data suggests some kind of parallel solution. Several processors could be set to work, each handling a part of the problem. But experiments in this direction tend to run into the same bottlenecks: memory access for instructions and data. Furthermore, the additional hardware and software overhead required to control and synchronize the processors may even slow the system down.

These bottlenecks are called the von Neumann bottlenecks: an unfitting tribute to the father of the digital computer. They can be stretched, but they can't be avoided. Computer theorists have begun looking outside the bottle. Two "non–von Neumann" strategies for getting out are *pipelining* and *data-flow architecture*.

NEC Electronics has developed a microprocessor chip based on these principles, the μPD7281. Designed primarily as an image-processing chip, the μPD7281 uses pipelining and data-flow architecture to allow processing of image data (such as enlarging, shrinking, and enhancing images) at high speed—5 million instructions per second (MIPS).

INTO THE PIPELINE

Pipelining increases throughput by using a processor's resources more fully. In a pipelined system, the processor starts working on the next step of the problem before it has finished with the last step. To some degree, many von Neumann–style processors use pipelining. For example, prefetch registers are a form of pipelining. At the same time as one instruction is being executed, the next instruction can be brought into the CPU from memory. Now, the additional time required to fetch an instruction is effectively zero; that operation takes place while the CPU is busy elsewhere.

The μPD7281 is thoroughly pipelined, inside and out. Inside, the μPD7281 is basically a pipeline with a loop in it (figure 1). The loop consists of several "blocks," or working areas. As soon as a block completes its work, it passes the results to the next block and takes more data from the previous block. The loop allows data to pass through the system as many times as necessary. Like an assembly line, each station is always busy; an individual product may take all day to be built, but the factory doesn't wait for one product to be finished before starting the next.

When μPD7281s are used together, they are arranged in a straight pipeline (figure 2). Data goes in one end and comes out the other. Each chip passes its output directly to the input of another chip. There is no interface hardware at all. Each μPD7281 has a module number, which identifies the particular data that it processes.

The delay between entering a set of figures and getting the answer for those figures depends on the problem. However, once the pump has been primed, if the pipeline is kept full (i.e., if you continuously input data), you can get an answer every 200

(continued)

Tom Jeffery is a technical writer for NEC *Electronics Inc.* (401 *Ellis St., Mountain View,* CA 94039).

nanoseconds (ns) at 10 megahertz (MHz).

Data-Flow Architecture

The second and really radical innovation for the μPD7281 is its data-flow architecture. Von Neumann architecture is program-driven and runs on more or less sequential instructions that manipulate data. Conversely, data-flow architecture is data-driven and runs on "tokens," which are packages containing both instructions and data. In the μPD7281, a token consists of 16 bits of data (plus sign and control bits) and a varying amount of identification (ID) and instructions. These packages flow into the μPD7281 and along the pipeline. They do not have to be fetched. In general, they can be input in any order because they contain their own ID and operational information. The processor performs operations when all the necessary data is present.

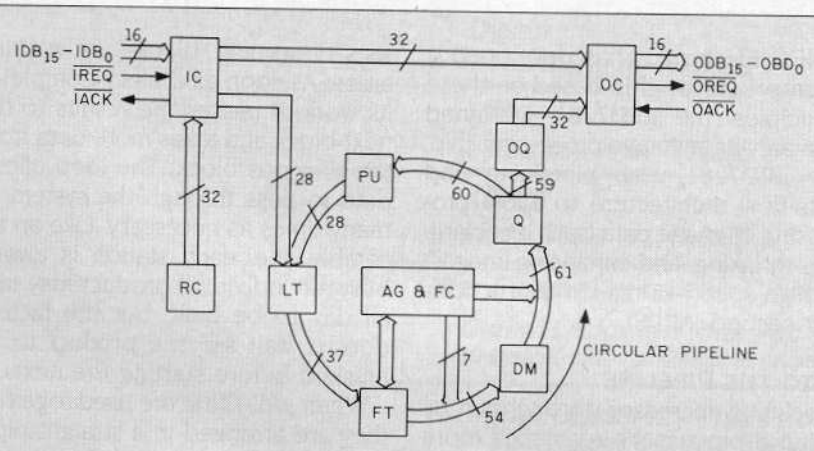
For a simplified example, think of adding two numbers. In most computers (von Neumann architecture), you write a program that says "Add A to B and put the result in C." To execute the program, the computer gets the first instruction, which tells it to get A. The next instruction tells it to get B. The next tells it to add them. The next tells it where to store the result. That's four instruction fetches: two data fetches, one addition, and one data store.

In a data-flow machine, your program says the same thing, "Add A to B and put the result in C." The host processor puts the A data in a token marked "A," puts B in a token marked "B," and sends these to the data-flow machine whenever it wants, in whatever order it wants. The answer will be output, labeled "C." No instruction fetches, no data fetches. And the pipeline spits out answers as fast as you can shove in the data.

Data-flow "programs" are easily represented as flow graphs, in the same way that conventional, sequential programs are represented by flowcharts. In a flow graph, the lines—"arcs" or "links" in mathematical terms—represent the flow of tokens, and the boxes, or "nodes," represent operations on the tokens. As you can see from figure 3, flow graphs express the nonsequential, parallel nature of the data-flow concept. Operations are performed as tokens arrive along links at a node. The order in which the operands arrive does not matter.

These concepts may be easier to explain by looking at the chip in more detail (refer to the block diagram of the μPD7281 in figure 1). The μPD7281 is connected to the outside world by two 16-bit buses, an input and an output bus. Behind these buses are an input controller and an output controller. The input controller puts two 16-bit words together into a single 32-bit token. It then checks the token to see if it is addressed to this μPD7281. Each μPD7281 has a 4-bit address or module number. Each input/output (I/O) token has a module number, the address of the μPD7281 it wants to go to. If the module number on the token is different from the module number of the present μPD7281, the input controller passes the token unchanged straight to the output controller. This takes only one clock cycle, making the μPD7281 practically transparent to a token that is not addressed to it. The input and output controllers are analogous to the doors of a hospital. The "In" door and the "Out" door are close together, so if you are in the wrong building, you can go right on to the next one.

The μPD7281 uses a two-line handshaking system. When a processor wants to output a token, it signals on its $\overline{\text{OREQ}}$ (output request) line. If it can accept, the receiving processor signals on its $\overline{\text{IACK}}$ (input acknowledge) line. The input controller uses



LEGEND

- IC: Input Controller. Controls input data tokens and determines whether or not an input data token should be sent to the circular pipeline for processing.
- OC: Output Controller. Controls output data tokens.
- LT: Link Table [128 words x 16 bits]. Stores instruction parameters.
- FT: Function Table [64 words x 40 bits]. Stores instruction parameters.
- DM: Data Memory [512 words x 18 bits]. Stores constants or temporary data.
- Q: Queue [48 words x 60 bits]. FIFO queue. Data Queue: 32 words x 60 bits. Generator Queue: 16 words x 60 bits.
- PU: Processing Unit. Executes logical, arithmetic, and bit operations.
- OQ: Output Queue [8 words x 32 bits]. FIFO queue for the output tokens.
- AG&FC: Address Generator and Flow Controller. Generates addresses for DM and controls the flow of tokens.
- RC: Refresh Controller. Generates refresh tokens for internal DRAMs.

Figure 1: *The internal architecture of the μPD7281, a pipeline with several blocks, or working areas, in a loop. The numbers represent the bit width of the path between blocks.*

the handshake signals to control access to the μPD7281. If the internal pipeline is full, it holds back until there is a place for a new token.

Tokens that are accepted by the input controller are stripped of their module number, now unnecessary. A token undergoes many format changes as it passes through the μPD7281, as you can see by looking at the changing width of the pipeline bus and the token formats in figure 4.

The first block in the pipeline is the link table, a 128- by 16-bit RAM (random-access read/write memory). The contents of the link table are downloaded from the host system. They are part of the μPD7281 "program"; specifically, they represent the links in the data flow graph. The ID field of the token addresses a location in the link table. The contents of that location are an address in the function table. This address, a new ID, and a few stray bits of code are attached to the data, and the new token is clocked into the next block, the function table.

The function table is a 64- by 40-bit RAM. The contents, accessed by the function-table address from the link table, are the other part of the μPD7281 program. They represent the nodes of the flow graph. Here the token picks up 40 bits of instruction

(continued)

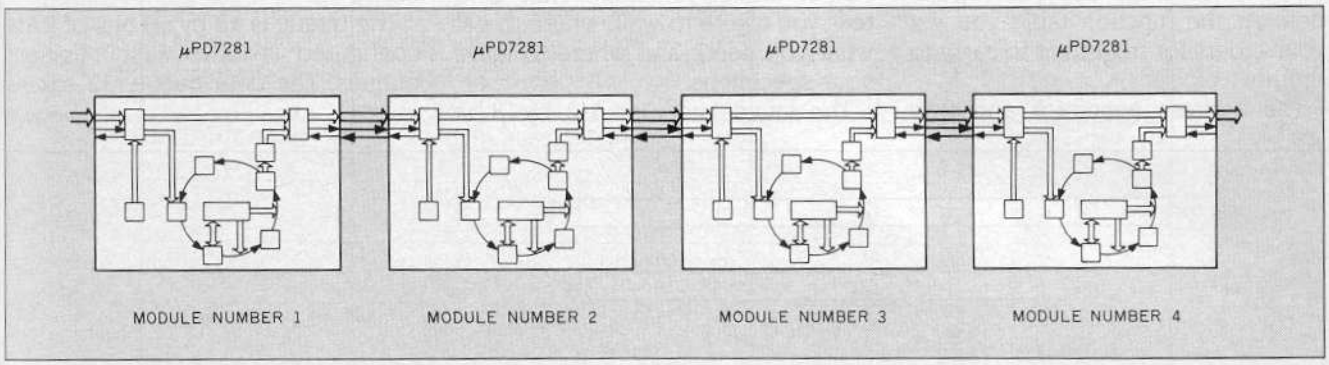


Figure 2: *When several μPD7281 chips, each with an internal pipeline, are connected together, they form a larger pipeline. Data, in the form of tokens, flows into the input of the first μPD7281, circulates through the internal pipeline, then passes out to the next μPD7281. This four-stage pipeline may work on up to 28 different tokens at a time.*

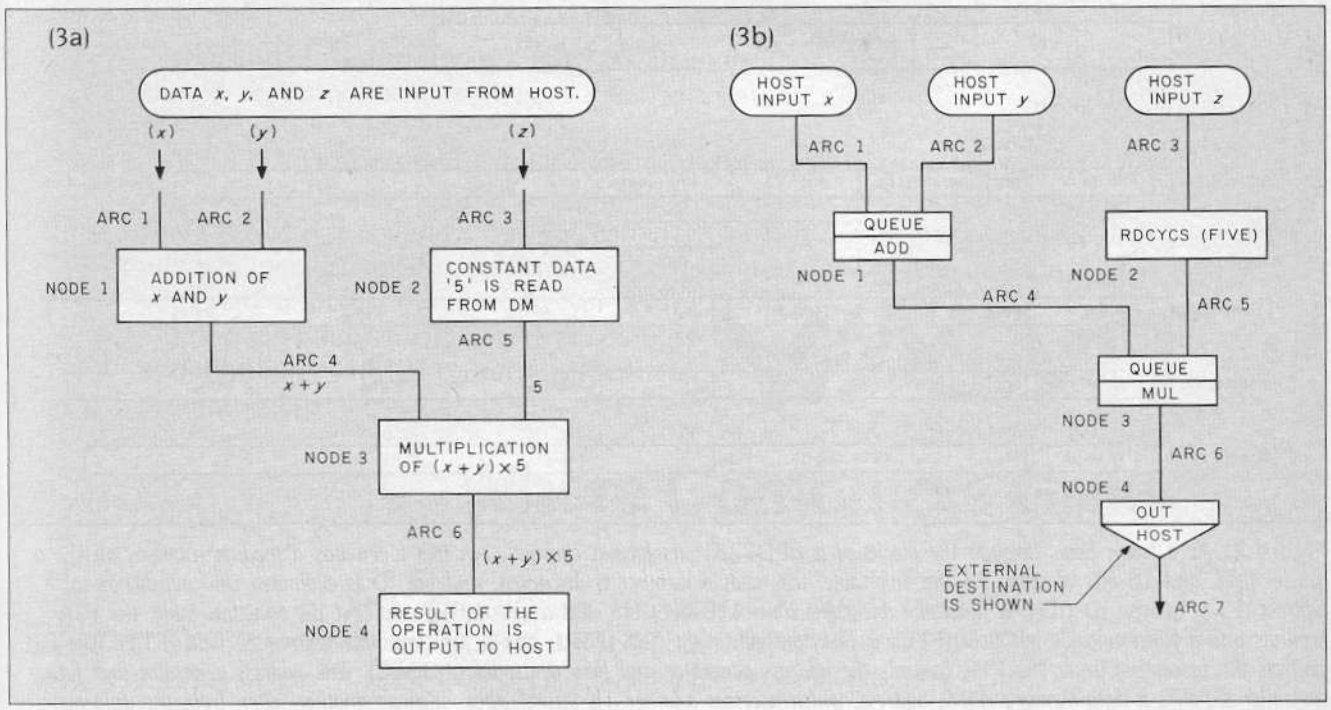


Figure 3: A *flow graph (3a) of the operation 5(x + y) as shown in 3b. When more than one token is expected at a node (as in* NODE1 *and* NODE3*), the node queues the first token until the second token arrives.*

codes, in the form of a function-table left field (processing-unit code), function-table right field (address-generator-and-flow-controller code), and function-table temporary field (counters and miscellaneous).

To continue the hospital analogy, the link table is akin to the receptionist. (Picture yourself in the role of patient-token.) It takes your ID and tells you who to see in the function table. The function table acts as the doctor who prescribes your operation. At the function table you get your orders for treatment to carry to the next blocks.

The address generator and flow controller comes into play if the operation to be performed, as determined by the function-table right field, involves more than one token. It acts as the back office, the secretary who knows where everything is. This block allows tokens to read from or write to sequential blocks of data memory (address-generator functions). And it queues tokens that are waiting for a second operand to perform their operation (address-generator-and-flow-controller function). That is, it tells you where to wait, where to get what you need, and where to leave your specimens.

The data memory is a 512- by 18-bit RAM that holds the first operand of a two-operand function until the second arrives. It also acts as general-purpose temporary storage and an I/O buffer. It operates under the control of the address generator and flow controller. The data memory is the hospital's files. You pick up or leave data about your case here. The address generator and flow controller will keep track of it. Then you go to the queue. The queue is the waiting room.

The queue is 48 by 60 bits of RAM configured as two first-in/first-out queues. The data queue, 32 tokens

(continued)

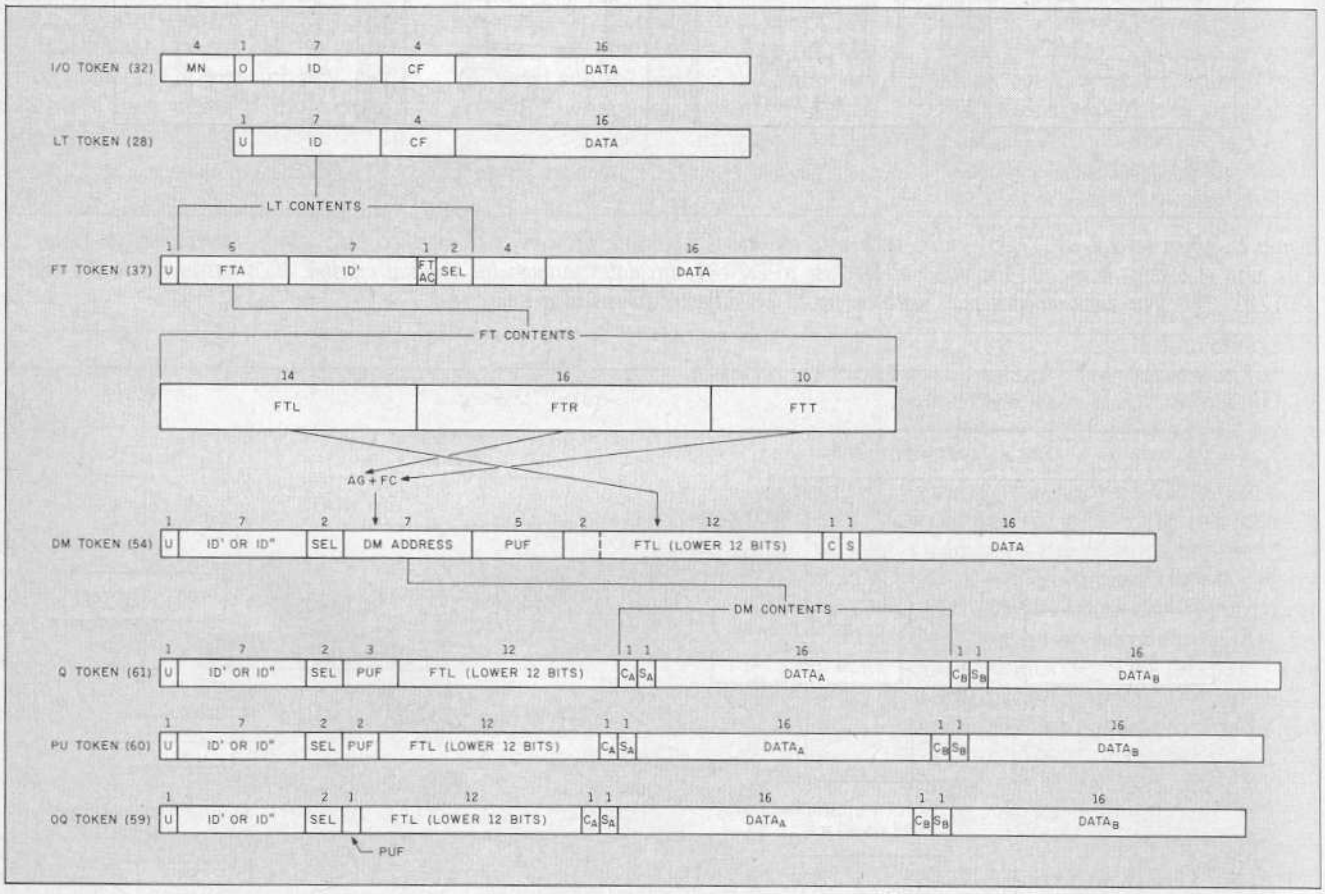


Figure 4: *As a token flows through the blocks of a μPD7281, its format changes. An I/O token has a module number, an ID, a control field, and 16 bits of data. At the link table, the module number is discarded, and the ID is expanded into a function-table address (FTA), a new ID (ID'), a function-table right control (FTRC) bit, and a select (SEL) bit. At the function table, the FTA expands into a function-table left field (FTL), a function-table right field (FTR), and a function-table temporary field (FTT). The FTL controls the processing unit. The FTR controls the address generator and flow controller (AG&FC). The address generator and flow controller supplies a data-memory (DM) address, which accesses another 18 bits of data, such as another token from the data-memory queue. The processing-unit field (PUF) specifies either a refresh token (in a queue token), a read/write token (in a processing-unit token), or neither (in an output queue token), and decreases accordingly in size.*

long, holds tokens bound for the processing unit or the output. The 16-token generator queue holds tokens to be copied. The queues provide slack in the pipeline. Since the processing unit takes longer to process some instructions than others, the queues are necessary to keep slower operations from excessively backing up the processor pipeline. They do not release tokens to the output queue or the processing unit until these blocks are free. A subtle algorithm controls their operation. The data queue has priority over the generator queue if there are eight or more tokens in the data queue. This restricts the generator queue tokens, which are more dangerous because they in turn create more and more tokens and thus have the greater potential for causing pipeline overflow. The data queue has a restrict/inhibit mode to inhibit the input controller from accepting new tokens when the data queue has more than 24 tokens.

When one or two tokens are ready in the queue, they get passed to the processing unit. This is like the ALU (arithmetic and logic unit) of a von Neumann microprocessor, with a very rich instruction set. It performs the usual logic, arithmetic, and compare functions, including the 200-ns 17-bit signed multiply or divide, and many extras, such as a barrel shift (1- to 16-bit shift in one 200-ns cycle) and double-precision adjust. It also performs the token-generating functions, which make copies of a token. These operations are specified by the contents of the function-table left field. Since they are specified separately, a token can have both address-generator-and-flow-controller instructions and processing-unit instructions. These instructions are listed in table 1.

After your operation in the processing unit, you go back into the pipeline at the link table. You now have a new ID, specifying a new course of treatment. You keep going around until the link-table contents specify a function-table address whose contents in turn call for the token to be output, that

(continued)

Table 1: *The μPD7281's instruction set. A single token may include address-generator-and-flow-controller instructions and processing-unit instructions.*

Address-Generator-and-Flow-Controller Instructions

Mnemonic	Instruction
QUEUE	queue
RDCYCS	read cyclic short
RDCYCL	read cyclic long
WRCYCS	write cyclic short
WRCYCL	write cyclic long
RDWR	read/write data memory
RDIDX	read data memory with index
PICKUP	pick up data stream
COUNT	count data stream
CONVO	convolve
CNTGE	count generation
DIVCYC	divide cyclic
DIV	divide
DIST	distribute
SAVE	save ID
CUT	cut data stream

Processing-Unit Instructions

Mnemonic	Instruction
OR	logical OR
AND	logical AND
XOR	logical EXCLUSIVE-OR
ANDNOT	logical INVERT first operand then AND [A AND B]
NOT	invert
ADD	add
SUB	subtract
MUL	multiply
NOP	no operation
ADDSC	add, shift, and count
SUBSC	subtract, shift, and count
MULSC	multiply, shift, and count
INC	increment
DEC	decrement
SHR	shift right
SHL	shift left
SHRBV	shift right with bit reverse
SHLBRV	shift left with bit reverse
CMPNOM	compare and normalize
CMP	compare
CMPXCH	compare and exchange
GET1	get one bit
SET1	get one bit
CLR1	clear one bit
ANDMSK	mask a word with logical AND
ORMSK	mask a word with logical OR
CVT2AB	convert two's-complement to sign-magnitude
CVTAB2	convert sign-magnitude to two's-complement
ADJL	adjust long (for double-precision numbers)
ACC	accumulate
COPC	copy control bit

Generate Instructions

Mnemonic	Instruction
COPYBK	copy block
COPYM	copy multiple

(continued)

SETCTL	set control field)

Output Instructions

Mnemonic	Instruction
OUT1	output 1 token
OUT2	output 2 tokens

Listing 1: *Assembly-language listing of the flow graph in figure 3. Lines 1 through 4 constitute the declaration field; lines 5 through 15, the instruction field.*

```
1    EQUATE      HOST = 0;
2    MODULE      EXONE = 8;
3    INPUT       ARC1, ARC2, ARC3;
4    OUTPUT      ARC7;

5    LINK        ARC4 = NODE1 (ARC1, ARC2);
6    LINK        ARC5 = NODE2 (ARC3      );
7    LINK        ARC6 = NODE3 (ARC4, ARC5);
8    LINK        ARC7 = NODE4 (ARC6,     );

9    FUNCTION    NODE1 = ADD, QUEUE (QUE1, 1);
10   FUNCTION    NODE2 = RDCYCS     (FIVE,  1);
11   FUNCTION    NODE3 = MUL (Y), QUEUE (QUE2, 1);
12   FUNCTION    NODE4 = OUT1       (HOST, 0);

13   MEMORY      QUE1 = AREA (1);
14   MEMORY      QUE2 = AREA (1);
15   MEMORY      FIVE = 5;
```

is, until your new ID says you are cured. At this time, a token goes to the output queue from the queue, instead of to the processing unit. The token is then fitted up with its module number and ID, and it leaves the μPD7281, in exactly the same format as it entered, via the output controller.

Because the output tokens have the same format as the input tokens, interfacing μPD7281 to μPD7281 is simplicity itself. The 18 output lines from one processor go to the 18 input lines of the other (ODB_0 through ODB_{15} to IDB_0 through IDB_{15}, $\overline{OREQ}$ goes to $\overline{IREQ}$, and $\overline{OACK}$ goes to $\overline{IACK}$). The only additional hardware required is a 4-bit module-number register for each μPD7281 to set the module number at RESET.

The interface to memory is a little more difficult. The concept of external memory addresses is alien to data-flow architectures. The configuration shown in figure 5 requires a memory-bus interface to deal with the world outside the pipeline. This arrangement makes the memory look like a special kind of μPD7281 that accepts tokens addressed to it and passes others back to the beginning of the pipeline. Now the pipeline has looped back on itself, like the internal pipelines. Wheels are thus contained within wheels. To the memory, the interface should look more like a DMA (direct memory access) controller. It could also control a display processor and handle the host CPU interface. NEC has plans for just such a memory interface, called the MAGIC (memory address generator and interface controller) chip.

The μPD7281 pipeline is not designed to operate independently. It requires a host system, but it is designed to run somewhat loosely linked to the host. Nominally, the host is required to download the software into the link and function tables and to start the processing. It should be able to monitor the results, probably with an interrupt system. Although we haven't discussed error handling, the host system can read the system state on an error condition. These concepts are built into the μPD7281 and are easily implemented.

Now that we have our multiprocessor system, composed of a host, memory, and a few μPD7281s, how do we program the μPD7281s? What language do we use? No high-level language is suited to deal with this kind of architecture. NEC has no equivalent of Occam, the language for INMOS's multiprocessing Transputer. The μPD7281 assembler, however, is really quite simple.

The assembler is based on the flow-graph concept. Listing 1 shows an assembly-language program based on the flow graph in figure 3. The EQUATE statement simply assigns a constant to a variable; in this case, the host address to 0. The MODULE statement assigns this section of code to a given μPD7281. The input tokens (ARC1, ARC2, and ARC3) and output token (ARC7) are declared in the INPUT and OUTPUT statements. Look at the flow graph and you can see the connection.

The LINK statements show what node (or function) every arc in the graph comes from and which arcs went into the node. For instance, the statement LINK ARC4 = NODE1 (ARC1, ARC2); means that the link ARC4 is the result of the the function NODE1 operating on links ARC1 and ARC2.

LINK statements generate link-table entries. The statement above puts the address of function NODE1 as the function-table address and the address of ARC4 in the ID' section of link-table addresses ARC1 and ARC2.

The FUNCTION statements show what instructions make up each function on the flow graph. So FUNCTION NODE1 = ADD, QUEUE (QUE1, 1) means that the node NODE1 performs an addition and uses the queue QUE1 to hold whichever operand comes first until the second one ar-

(continued)

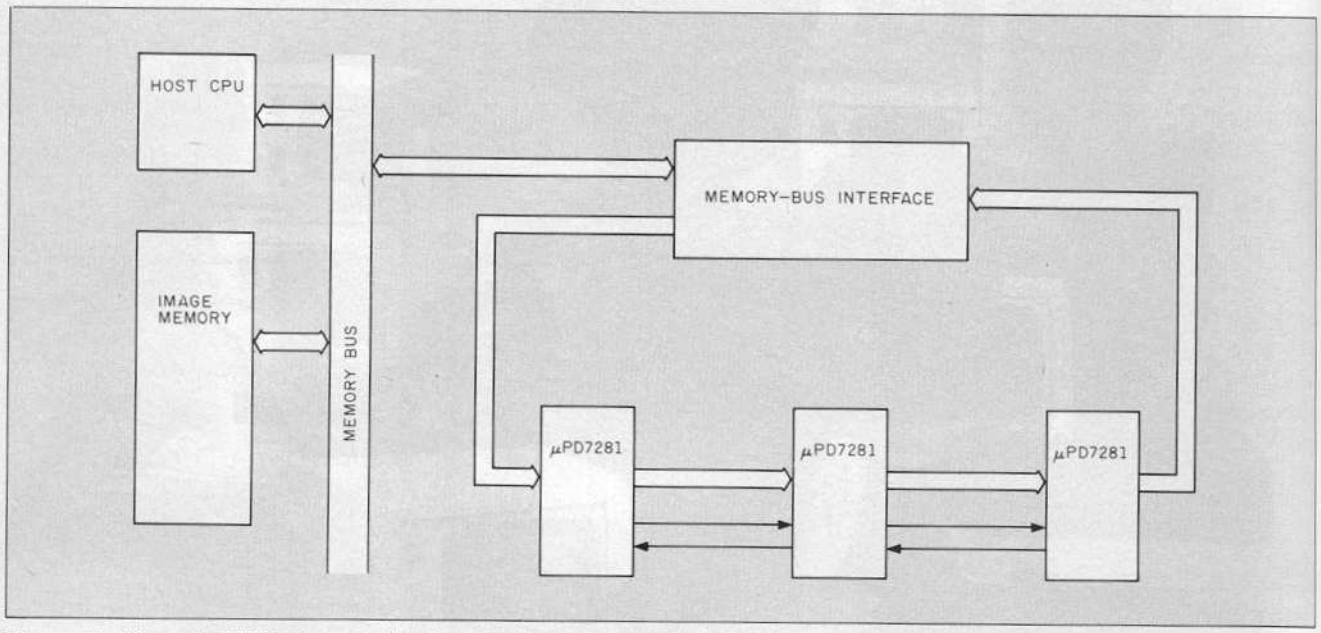


Figure 5: *Three μPD7281s in a pipeline loop with a memory-bus interface to external memory.*

rives at the node.

These statements become function-table entries. The ADD instruction goes in the function-table left field, and the QUEUE instruction goes in the function-table right field.

Data memory is allocated by the MEMORY statements. The area QUE1 is defined as an area to queue one operand. The location FIVE is set to 5.

That's pretty much it. The code is not particularly readable, of course. It looks a little like a string of declarations, and you wonder where the action is. The tokens supply that. The problem is that the program is a one-dimensional listing of the flow graph. Once you have the flow graph, the assembler is a snap, but no one would think of writing the assembler first, in the way that you might write a BASIC program without a flowchart.

Writing a program for more than one processor is a little more difficult. When I first comprehended the idea, I thought, "To program like that, you'd have to have more than one brain."

First of all, you must break the problem up between the processors. There are two ways to do this. One way is to break up the data. For example, in image processing, a 512- by 512-pixel image could be broken up into four quadrants of 256 by 256 pixels. Each quadrant would be worked on by a separate μPD7281, each with the same program. Since the processors are working independently, the processing time will be cut to a quarter of the time required by one μPD7281.

The other choice is to partition the algorithm. Since the link table, function table, and data memory are small, it isn't hard to imagine a program too large (more than 128 links and 64 functions) to fit on one μPD7281. A program of this size must be split up and put on different chips. For example, one μPD7281 shifts images, one sizes them, and one rotates them.

Routine partitioning is a little more tricky than data partitioning. The processes on each chip must be fairly well balanced, or the most heavily loaded chip will slow down the system. Of course, in the data-partition method the data must be partitioned equally. But if you are going to partition your program, you should do a simulation to ensure the balance.

This isn't the only problem you can run into when partitioning a problem among processors. When an early chip's routines run faster than a later chip, the tokens can pile up in the slower chip's data memory. This should be handled by routine balancing, but if it can't be avoided, a synchronizing token can be passed by the later μPD7281 back to the earlier to start and stop it.

Are there limits to the speed to be gained by adding processors? Of course. The fact that there are only 14 valid module numbers is one limit. Another is the memory bus. When a routine performs simple operations on many pieces of data, the memory bus is being used more than the μPD7281s. Adding more processors will fail to increase processing speed. This results in a state known as "bus neck." The system becomes memory-bus-bound, just like a von Neumann machine. The way to find the maximum number of effective processors is through simulation.

New software design techniques for this chip remain to be worked out. The most obvious would be a graphic assembler that codes directly from a flow graph.

It may even be possible to make a C or Pascal compiler for the μPD7281 that would translate your algorithm into the data-flow domain and optimize it for the best number of chips. ■