

# Flexible Microprocessor Register File

## BACKGROUND AND SUMMARY

The present application relates to programmable circuits, and more particularly to I/O circuitry with selectable data reordering for graphics.

A vector processor or array processor is a CPU design that is able to run mathematical operations on multiple data elements simultaneously. A serial vector is a sequence of data held in registers that are processed by the same instruction. For example, a single instruction may cause four registers to be added to another four and the result written to a further four. A parallel vector holds several data items within the same register, each of which has the same instruction applied to it. Vector processing improves code density and allows optimizations that improve performance.

A common problem suffered by vector processors is the need to organize data within the register file such that the same instruction may be applied to a series of registers. Register files generally only allow simultaneous access to a set of values aligned along a particular direction, i.e., along a row of the vector. Accordingly, a single instruction can access multiple values for a horizontal operation, but vertical operation requires either transposing the array being operated or performing separate access operations for each value in a different row. It is common to spend several instructions re-arranging data to make it suitable for vector processing and this overhead may obviate the benefits of using a vector.

In view of these limitations, more efficient architectures and methods for performing transpose and other array manipulations are desired.

Yet another problem arises when a program instruction indirectly accesses a register. Microprocessors control programs' access to register files. Because of pipelining, some instructions must be stalled until the register from which they will read has been written to by another instruction. Scoreboarding stalls these

instructions, so the program need not manage stalling. Stall condition is usually applied early in the execution pipeline. However, if a register is to be accessed indirectly by a program instruction, the register may not be known until it is too late--until after the stall condition would normally have already been applied. Without knowing the register at that earlier time, it is difficult to apply stall conditions for instructions that use indirect access.

The inventions disclosed in the present application provide mechanisms to handle indirect register access without additional scoreboarding hardware, and can be further used to build a flexible FIFO access mechanism.

## **Flexible Register File I/O Architecture**

The present application discloses a register file input/output configuration in which a variety of data transpositions are available at minimum power. Power is conserved by avoiding register-to-register data transfers; instead, the sequencer provides executable microinstructions which imply a variety of apparent data formats (as seen by the data channel), without unnecessary physical transfers of data.

Various disclosed embodiments provide new ways for microprocessor register-files to be accessible, in multiple formats in order to reduce the number of program instructions required during byte, word and long word data reformatting. The disclosed innovations, in various embodiments, provide one or more of at least the following advantages:

- Variety of data rearrangements;
- Minimal power consumption;
- Easy accommodation to special data reordering for digital signal processing operations;
- Suitability to customized access to data with two-dimensional structure;
- Suitability to customized access to data with multidimensional structure.

### **BRIEF DESCRIPTION OF THE DRAWINGS**

The disclosed inventions will be described with reference to the accompanying drawings, which show important sample embodiments of the invention and which are incorporated in the specification hereof by reference, wherein:

**Figure 1** shows how four consecutive registers are viewed with byte-transpose enabled. Each row in the diagram represents one register as viewed by a program. When Byte-Transpose is enabled, the register file is effectively rotated by 90°, so that Register 0 contains all the low-bytes of the four registers, register 1 contains all the second-bytes of the four registers, and so on.

**Figure 2** shows how two consecutive registers are viewed with word-transpose enabled. Each row in the diagram represents one register as viewed by a program. When Word-Transpose is enabled, the register file is effectively rotated by 90°, so that Register 0 contains all the low-words of the two registers, register 1 contains all the high-words of the two registers

**Figure 3** shows the data in register 0 being byte swapped in two different ways. The first is a full (DCBA) byte-swap, in which the original data-bytes are swapped within the entire 32-bit word, and the second shows a BADC byte-swap taking place, which swaps the bytes within each word.

Figures 5a-5g are a set of related drawings. **Figure 5a** shows a sample hardware register configuration, in which the register is separated into multiple multiport RAMs, each having multiplexers connected to each of its data lanes. Figures 5b-5g show different states of operation of this register: **Figure 5b** shows the routing needed for a 32-bit word at address 0 without transpose; **Figure 5c** shows routing for address 1 without transpose; **Figure 5d** shows the routing needed for the first 32 bits of an eight bit transpose; **Figure 5e** shows the routing for the second 32 bits of an eight bit transpose; **Figure 5f** shows the routing for

address 0 with a 16-bit transpose, in this sample implementation; and **Figure 5g** shows routing for address 1 with a 16-bit transpose, in this sample implementation.

## **DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS**

The numerous innovative teachings of the present application will be described with particular reference to the presently preferred embodiment (by way of example, and not of limitation).

### **Transposable Register-File Operation**

The transposable register-file is a novel microprocessor register-file data organization scheme which overcomes many of the disadvantages of traditional data organization in microprocessor register-file, and which has the benefits of allowing a microprocessor register-file to be viewed in multiple formats with a reduction of the number of program instructions required during byte, word and long word data reformatting. The preferred embodiment supports both byte-transpose and word-transpose.

#### **Byte-Transpose Register File**

Figure 1 shows how four consecutive registers are viewed with byte-transpose enabled. With reference to Figure 1, left hand side (110) illustrates those registers before transpose enabled and right hand side (120) illustrates the same registers after transpose enabled. Each row in Figure 1 represents one register as viewed by a program. For instance, bottom row 211 shows Register 0 before transpose enabled. Each register in turn is composed of four bytes with the left most (for instance 0a) being the lowest byte and the right most (for instance 0d) being the highest byte. When byte-transpose is enabled, the register file is effectively rotated by 90°, so that Register 0 (121) contains all the low-bytes of the four registers, Register 1 (122) contains all the second-bytes of the four registers, and so on.

### **Word-Transpose Register File**

Word-transpose is similar to byte-transpose, except that the register data is rotated on a per word basis instead of per byte basis. **Figure 2** shows how two consecutive registers are viewed with word-transpose enabled. With reference to **Figure 2**, left hand side (210) illustrates those registers before transpose enabled and right hand side (220) illustrates the same registers after transpose enabled. Each row in **Figure 2** represents one register as viewed by a program. For instance, bottom row (111) shows Register 0 before transpose enabled. Each register in turn is composed of two words with the left most (for instance 0a) being the low word and the right most (for instance 0b) being the high word. When word-transpose is enabled, the register file is effectively rotated by 90°, so that Register 0 (221) contains all the low-words of the four registers, Register 1 (222) contains all the high-words of the two registers.

### **Register-File Byte-Mapping and Byte-Masking**

The register-file byte-mapping and byte-masking functions add further flexibility to the novel microprocessor register-file data organization scheme. This feature of the disclosed inventions allows a program to arbitrarily reorganize the bytes within a register and has the benefits of further reduction of the number of program instructions required during byte, word and long word data reformatting.

#### **Register-File Byte-Mapping**

Byte-Mapping allows a program to arbitrarily reorganize the bytes within a register in order to isolate, or group, interesting sub-components when reading from, or writing to the register-file. **Figure 3** shows two examples of byte-mapping on a register. With reference to **Figure 3**, left hand side (310) illustrates those registers before byte-mapping and right hand side (320) illustrates the same

registers after byte-mapping. Each row in **Figure 3** represents one register as viewed by a program. For instance, bottom row (311) shows the Register before byte-mapping. Each register in turn is composed of four bytes with the left most (for instance 0a) being the lowest byte and the right most (for instance 0b) being the highest byte. When a byte-mapping of full (DCBA) byte swap is enabled, the original data-bytes are swapped within the entire 32-bit word, and the bytes in register (312) are reorganized as bytes in register (322). When a byte-mapping of (BADC) byte swap is enabled, the original data-bytes are swapped within each word, and the bytes in register (311) are reorganized as bytes in register (321).

#### **Register-File Byte-Masking**

The preferred embodiment supports both byte-mapping and byte-masking. Register-file byte masking is another novel microprocessor register-file data organization scheme that provides control over the bytes that are modified by an instruction in order to accelerate insertion of data into existing register. The program may specify a byte-mask both for source operands and destination operands. When byte-mask is specified for source operands, parts of a register may be forced to zero on input to an instruction. When byte mask is specified for destination operands, the result of an instruction can be written to parts of a destination register.

#### **Indirect Register Access**

The indirect register access has the benefits of providing indirect register access without additional scoreboarding hardware. It provides two types of instructions: one for moving data from one register to another register, and another for synchronization.

The instruction format for moving data specifies the following parameters: a register that holds the source data, a register that holds either the destination register or the index of the destination register, and optionally a count of the number of registers to transfer. If the destination register is directly referenced in the instruction, those registers directly referenced in the instruction are scoreboarded when the instruction is executed. However, if the destination register is not directly referenced in the instruction, those registers indirectly referenced in the instruction are not scoreboarded when the instruction is executed and synchronization instruction will be used to ensure that the data in the register indirectly accessed is correct.

In a typical use of this invention, a programmer uses a number of registers as scratchpad memory. Data is loaded into the scratchpad. If there is a switch from a direct to indirect access of register or vice versa, a synchronization instruction is issued to calculate an index into the scratchpad and the contents of the register at that index are copied into a known register. At this point all processing elements may use the same instruction to process data at the same register index. When the calculation is complete, the result may be copied back to the scratchpad and another synchronization instruction is issued to calculate the index.

## **Implementation of Hardware Data FIFO in Register-File**

The provision of hardware data FIFO in microprocessor register-file uses similar ideas of indirect register access. This innovative feature, in the preferred embodiment, sets aside a number of registers from the microprocessor register-file for the FIFO storage, and provides a mechanism for moving data into the FIFO, from another source, and for moving data serially out of the FIFO into other registers within the microprocessor. It has the benefits of:

- Building the FIFO in the processor register file allows those registers to be re-used as normal registers when the FIFO is not needed.
- The invention allows the size of the FIFO (and thereby the number of reserved registers) to be changed under program control.
- It solved the indirect access problem in a hardware register FIFO implementation.

## **Example of use of Transpose and Byte-Mapping**

Pixel data is often stored in what is called the RGBA8888 format, in which each pixel is made up of red, green, blue, and alpha components, each of 8 bits. All four components are packed into one 32-bit word for convenience of display.

In common algorithms such as blending the alpha component is used to modify the color components as follows:

$$\text{dstR} = (\text{srcR} * \text{srcA}) + \text{dstR}$$

$$\text{dstG} = (\text{srcG} * \text{srcA}) + \text{dstG}$$

$$\text{dstB} = (\text{srcB} * \text{srcA}) + \text{dstB}$$

Sample assembler code for this algorithm is:

```
mul tmp[0], src[0], src[3]
mul tmp[1], src[1], src[3]
mul tmp[2], src[2], src[3]
add dst[0], tmp[0], dst[0]
add dst[1], tmp[1], dst[1]
add dst[2], tmp[2], dst[2]
```

Where the syntax is *instruction, destination, source A, source B*. The array indices refer to the byte position in the pixel.

The code may be reduced if a parallel vector is used, but the alpha component must be repeated in each byte of a 32-bit register. This can be done using a byte swap mode:

```
set byte swap mode for srcB to DDDD
mul tmp, src, src
reset byte swap mode for srcB to ABCD
add dst, dst, tmp
```

Note that this code only produces 3 bytes of results even though the registers hold 4 bytes. If 4 pixels are processed as a serial vector this inefficiency can be removed:

```
transpose srcA
transpose srcB
vector_3_mul tmp, src, src
transpose dst
vector_3_add dst, dst, tmp
```

Transposing *srcA* causes all the red components to be in one register, all the green in another, and all the blue in a third. Transposing *srcB* causes all the alpha components to be in one register. Vector instruction of length three cause four pixels to be processed in 3 instructions (the stride of the *srcB* vector must be zero to use the same alpha value for each component).

### **Register File Implementation**

Details of a sample implementation will now be described. In this implementation, the register file is used for all storage within the processing element and holds a generous 256 registers, each 32-bits wide. The registers are perhaps more important to overall system performance than the ALU because they control the movement of data, and a SIMD array typically has high compute performance relative to data bandwidth. The register file can be large because it absorbs a number of FIFOs that would normally be needed to feed the ALU. All registers are preferably scoreboarded, so any instruction that attempts to read a register that has a write scheduled for it will stall until the write completes.

### **Parallel Vectors**

To make good use of the ALU, several data items may be packed into one register. The ALU may work on four 8-bit items at a time, or two 16-bit items, but the operation is always the same. This is similar to vector calculations, and when more than one item of data is held in a register it is referred to as a parallel vector (pvec as opposed to svec for vectors executed sequentially). Pvecs can boost performance if it is not too expensive to get data into an appropriate format.

An example of using pvecs is to take four pixels of red, green, blue, and alpha, and re-group them such that common components are in the same register (so grouped as RRRR, GGGG, BBBB, AAAA). Then different operations can be applied to each component at full speed (it is common for alpha to be processed differently than RGB). If you imagine the four pixels as a four by four array of bytes, the source format has RGBA in rows and the processing needs them in columns and to get into this format requires transposing the pvecs. After processing is complete the transpose needs to be reversed.

The register file supports zero-cost transposing for 8 or 16 bit pvecs. If the data type is 16 bits the register set is treated as being in pairs and the transposition takes place assuming two registers hold a 2x2 array of data. If the data type is 8 bits then four registers are assumed to hold a 4x4 array of data. Transposition is free because the register file is made up of four separate RAMs, which gives access to four different registers at the same time. The lower bits of the register address select the bytes to use, so registers to be transposed must be in sequential registers and must be aligned to the number of registers that will be transposed.

Transposition also allows efficient memory access for 24 bit components. If data is stored byte-planar with four bytes of each component stored in the same 32 bit word the layout would be as shown in Figure 4. This is a useful way to store 24 bit data because there is no wastage but neither is there a difficult address calculation or nasty data shifting. In some algorithms it is convenient to process the components individually, but in others the whole pixel may be needed. Transposition allows this byte planar format to be converted into 32 bit pixels.

The register file has, in principle, three read ports and two write ports. Two of the read ports are used by the ALU, as is one of the write ports. The remaining read and write ports are used to get memory data in and out of the registers. For best performance the RAM used to build the register file should have all five ports,

but that will make it large. A compromise is possible in which one read and one write port are removed.

Because the register file is made up of four separate RAMs for transposition, it is possible to arrange accesses to them so that while the ALU accesses one RAM another can be used for memory data. The vector operations result in the registers being accessed in a predictable pattern. The trick is to arrange the addressing so that memory accesses follow the same pattern as vector operations, but staggered so that they don't use the same RAM at the same time. This is not always possible when transposing because the ALU may need access to all four RAMs. When there is contention for the register file the memory wins and the ALU stalls (this is the cost of not having all 5 ports).

### **Indirect Accesses**

Indirect register access allows the contents of one register to form the index to another. It is obviously useful for histograms, but also for FFT data shuffling and median filtering. It is difficult to implement because all PEs may access different registers, which breaks the SIMD model and requires additional scoreboarding hardware.

The media processor imposes a slight restriction that avoids the hardware cost. Special instructions are used to copy data from one register to another; the register to copy from (or to) is specified in another register. The restriction is that while indirection is in use any register that may be indirectly accessed must not be used directly. This removes the need to scoreboard the indirectly accessed register, while the directly accessed register is scoreboarded to ensure correct operation. The cost is an extra instruction per indirection.

## **Details of Sample Hardware Implementation**

Figures 5a-5g are a set of related drawings, which collectively show a sample hardware implementation and its various operational modes.

**Figure 5a** shows a sample hardware register configuration, in which the register is separated into multiple multiport RAMs **510**, each having multiplexers **520** connected to each of its data lanes. Four RAMs may be connected to support transposing. Each RAM is 32 bits wide and shows four byte-wide lanes. Each RAM holds every fourth entry in the register file. The dotted boxes are multiplexers that switch between the two inputs. This hardware implementation permits all of the above functional relationships to be realized.

The multiplexers can be, for example, simple by-8 circuits having two states, selected by a single control bit (per multiplexer). These control bits can be set, for example, by appropriate configuration instructions.

Figures 5b-5g show different states of operation of this register. In these diagrams, only the active inputs to active multiplexers **520** are shown.

**Figure 5b** shows the routing needed for a 32-bit word at address 0 without transpose, in this sample implementation.

**Figure 5c** shows routing for address 1 without transpose, in this sample implementation.

**Figure 5d** shows the routing needed for the first 32 bits of an eight bit transpose; the lower byte of each RAM is connected to a different byte lane, in this sample implementation.

**Figure 5e** shows the routing for the second 32 bits of an eight bit transpose, in this sample implementation.

**Figure 5f** shows the routing for address 0 with a 16-bit transpose, in this sample implementation.

**Figure 5g** shows routing for address 1 with a 16-bit transpose, in this sample implementation.

This hardware implementation can of course be varied, but this shows how an extremely versatile set of output reordering options can be achieved by multiplexing, WITHOUT unnecessary register-to-register transfers (which consume power).

Additional detail of the preferred implementation is shown in U.S. application number 11/536,483, which is hereby incorporated by reference in its entirety. This implementation is an advantageous context for the disclosed inventions, but it should be emphasized that the I/O architecture described in the present application can also be used in many other contexts.

According to a disclosed class of innovative embodiments, there is provided: A method of selectably transposing data accessed in a register, comprising the actions of: storing data in  $n$  memory segments, each having  $n$  data lanes at the output thereof; and selectably connecting each of  $n$  data bus segments to a respective one of said  $n^2$  data lanes; whereby a desired data transposition is provided at the time of register access without register-to-register transfers.

According to a disclosed class of innovative embodiments, there is provided: An electronic system, comprising: a logic unit; and at least one I/O register, comprising multiple memory segments each holding a respective fraction of a data set, said data set being distributed across said segments in a consistent pattern, and each said memory segment providing multiple lanes of data path; and multiple

multiplexers, each connected to connect a respective output bus segment to a respective data path of a respective one of said memory segments.

### **Modifications and Variations**

As will be recognized by those skilled in the art, the innovative concepts described in the present application can be modified and varied over a tremendous range of applications, and accordingly the scope of patented subject matter is not limited by any of the specific exemplary teachings given.

For example, the multiple access modes provided by the disclosed embodiments are particularly useful for graphics and image processing, they can also be especially useful for data which has internal 3-D or 4-D structure (e.g. a time series of voxel images). In such cases the capability for customized data transpositions can help with filtering and transformations.

For another example, a flexible register can optionally implement some but not all of the transpositions described above, and/or can implement additional transpositions besides those listed.

For another example, the disclosed hardware implementation uses byte-wide "lanes", but alternatively and less preferably a different fineness can be used. If fast nibble transpositions are desired, 8 RAMs could be used instead of four, with 8 lanes instead of four on each RAM, and 8 output busses instead of four. Note, however, that the number of multiplexers would quadruple if this were done.

For another alternative and less preferable example, more logic can be added into the multiplexers if desired. For instance, the multiplexers can be given additional states wherein the 8-bit output is not only connected to a selected input (or none), but wherein the bits of the input can be permuted, pairwise exchanged, complemented, ANDed, etc. Additional control bits would preferably be routed to the multiplexers in such cases.

None of the description in the present application should be read as implying that any particular element, step, or function is an essential element which must be

included in the claim scope: THE SCOPE OF PATENTED SUBJECT MATTER IS DEFINED ONLY BY THE ALLOWED CLAIMS. Moreover, none of these claims are intended to invoke paragraph six of 35 USC section 112 unless the exact words "means for" are followed by a participle.

The claims as filed are intended to be as comprehensive as possible, and NO subject matter is intentionally relinquished, dedicated, or abandoned.

**Claims**

1. A method of selectably transposing data accessed in a register, comprising the actions of:

storing data in four memory segments, each having four data lanes at the output thereof, wherein each said data lane carries 8 bits of data; and

selectably connecting each of four data bus segments to a respective one of sixteen data lanes by selecting four multiplexers activated of a total of thirty-four multiplexers, based on whether said desired data transposition is bitwise transposition or wordwise transposition;

whereby a desired data transposition is provided at the time of register access without register-to-register transfers,

wherein the multiplexers are arranged so that bitwise transposition and wordwise transposition are performed.