Garp: A MIPS Processor with a Reconfigurable Coprocessor

John R. Hauser and John Wawrzynek
University of California, Berkeley

Abstract

Typical reconfigurable machines exhibit shortcomings that make them less than ideal for general-purpose computing. The Garp Architecture combines reconfigurable hardware with a standard MIPS processor on the same die to retain the better features of both. Novel aspects of the architecture are presented, as well as a prototype software environment and preliminary performance results. Compared to an UltraSPARC, a Garp of similar technology could achieve speedups ranging from a factor of 2 to as high as a factor of 24 for some useful applications.

1 Introduction

In recent years, *reconfigurable hardware*—usually in the guise of *field-programmable gate arrays* (*FPGAs*)—has been touted as a new and better means of performing computation [1, 2, 3]. Promoters argue that FPGAs can be used to create fast, application-specific circuits for any problem. Impressive speedups have been documented for a number of tasks, including DNA sequence matching [4, 5], textual pattern searching [6], and RSA encryption [7], to name just a few.

Despite these successes, any computer built wholy out of FPGAs must overcome some obstacles:

- FPGA machines are rarely large enough to encode entire interesting programs all at once. Smaller configurations handling different pieces of a program must be swapped in over time. However, configuration time is too expensive for any configuration to be used only briefly and discarded. In real programs, much code is not repeated often enough to be worth loading into an FPGA.
- No circuit constructed with an FPGA can be as efficient as the same circuit in dedicated hardware. Standard functions like multiplications and floating-point operations are big and slow in an FPGA when compared to their counterparts in ordinary processors.
- Problems that are worth solving with FPGAs usually involve more data than can be kept in the FPGAs themselves. No standard model exists for attaching external memory to FPGAs. FPGA-based machines typically include ad hoc memory systems, designed specifically for the first application envisaged for the machine.

This work is supported in part by DARPA grant DABT63-C-0048, ONR grant N00014-92-J-1617, and NSF grant CDA 94-01156.

Authors' E-mail addresses: `jhauser@cs.berkeley.edu` and `johnw@cs.berkeley.edu`.

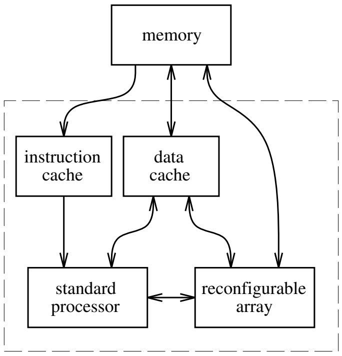


Figure 1: Basic Garp block diagram.

- Wide acceptance in the marketplace requires binary compatibility among a range of implementations. The current crop of FPGAs, on the other hand, must be reprogrammed for each new chip version, even within the same FPGA family.

To address some of these concerns, various researchers have proposed building a machine that tightly couples reconfigurable hardware with a conventional microprocessor [2, 8, 9]. The organization of such a hybrid, however, remains an open topic. In this paper we outline a candidate hybrid architecture, which we call *Garp*, in which the FPGA is recast as a slave computational unit located on the same die as the processor. The reconfigurable hardware is used to speed up what it can, while the main processor takes care of all other computation. Fig. 1 shows the organization of the machine at the highest level. Garp's reconfigurable hardware goes by the name of the *reconfigurable array*.

Garp has been designed to fit into an ordinary processing environment—one that includes structured programs, libraries, context switches, virtual memory, and multiple users. The main thread of control through a program is managed by the processor; and in fact programs need never use the reconfigurable hardware. It is expected, however, that for certain loops or subroutines, programs will switch temporarily to the reconfigurable array to obtain a speedup. With Garp, the loading and execution of configurations on the reconfigurable array is always under the control of a program running on the main processor.

Garp makes external storage accessible to the reconfigurable array by giving the array access to the standard memory hierarchy of the main processor. This also provides immediate memory consistency between array and processor. Furthermore, Garp has been defined to support strict binary compatibility among implementations, even for its reconfigurable

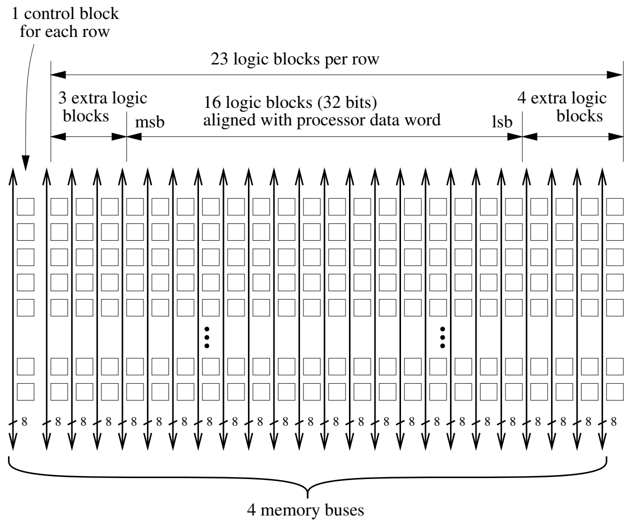


Figure 2: Array organization. In addition to the memory buses, a wire network carries signals between array blocks.

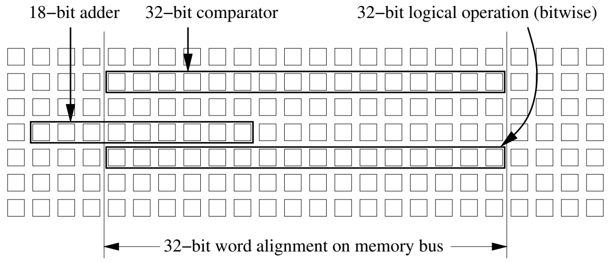


Figure 3: Typical natural layouts of multi-bit functions.

hardware.

Details of the array architecture are given in the next sections, followed by an overview of the programming environment and a look at likely speedups for a few real applications.

2 Garp Architecture[1]

Garp's main processor executes a MIPS-II instruction set extended for Garp. Any popular processor could have been used, but the MIPS is a common basis of research within the university community.

Garp's reconfigurable array is composed of entities called *blocks* (Fig. 2). One block on each row is known as a *control block*. The rest of the blocks in the array are *logic blocks*, which correspond roughly to the *CLBs* of the Xilinx 4000 series [10]. The Garp Architecture fixes the number of columns of blocks at 24. The number of rows is implementation-specific, but can be expected to be at least 32. The architecture is defined so that the number of rows can grow in an upward-compatible fashion.

The basic "quantum" of data within the array is 2 bits. Logic blocks operate on values as 2-bit units, and all wires are arranged in pairs to transmit 2-bit quantities. Operations on 32-bit quantities thus generally require 16 logic blocks. Multi-bit functions are naturally laid out along array rows (Fig. 3). With 23 logic blocks per row, there is space on each row for an operation of 32 bits, plus a few logic blocks to the left and right for overflow checking, rounding, control functions, wider data sizes, or whatever is needed.

Four *memory buses* run vertically through the rows for moving information into and out of the array. During array execution, the memory buses are used for data transfers to and from memory and/or the main processor. For memory accesses, transfers are restricted to the central portion of each memory bus, corresponding to the middle 16 logic blocks of each row. For loading configurations and for saving and restoring array state, the entire width of the memory buses is used.

The memory buses are not available for moving data between array blocks. Instead, a more conventional wire network provides interconnection within the array. Wires of various lengths run orthogonally vertically and horizontally. Vertical wires can be used to communicate between blocks in the same column, while horizontal wires can connect blocks in the same or adjacent rows. Unlike most FPGA designs, there are no connections from one wire to another except through a logic block. However, every logic block includes resources for potentially making one wire-to-wire connection independent of its other obligations.

The loading and execution of configurations is under the control of the main processor. Several instructions have been added to the MIPS-II instruction set for this purpose, including ones that allow the processor to move data between the array and the processor's own registers.

An individual configuration covers some number of complete rows of the array, which may be less than the total number of physical rows in the array. Distributed within the array is a cache of recently used configurations, so that programs can quickly switch between several configurations without the cost of reloading from memory each time. As with traditional memory caches, the size and management of the configuration cache is transparent to programs.

Data registers in the array are latched synchronously according to an *array clock*, whose frequency is fixed by the implementation. No relationship between the array clock and the main processor clock is required, although it is intended that the two clocks be the same. A *clock counter* governs array execution. While the clock counter is nonzero, it is decremented by 1 with each array clock cycle. When the clock counter is zero, updates of state in the array are stalled, effectively stopping the array. (Copies to the array by the main processor may still modify array state.) The main processor sets the array clock counter to nonzero to make the array execute for a specific number of array clock steps.

The *control blocks* at the end of every row serve as liaisons between the array and the outside world. Among other things, control blocks can interrupt the main processor and can initiate data memory accesses to and from the array.

The division of the array into rows to simplify array management is a technique that was first reported for the Dynamic Instruction Set Computer (DISC) [11]. Garp resembles DISC also in the way that multi-bit operations are naturally oriented across rows, and that global buses run orthogonally through the rows for bringing values into and out of the array.

[1]A complete reference manual will be available on the Web at `http://http.cs.berkeley.edu/projects/brass/garp.html`.

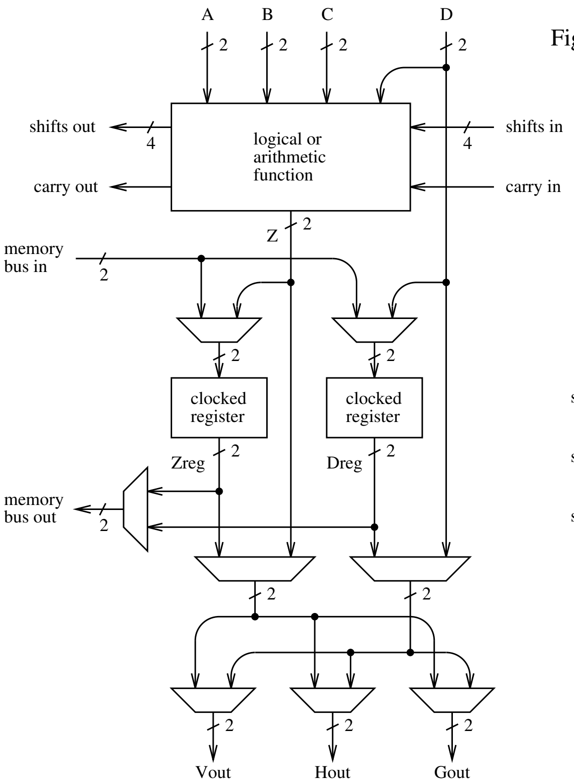
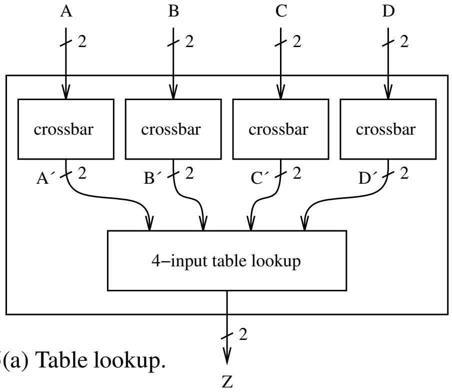
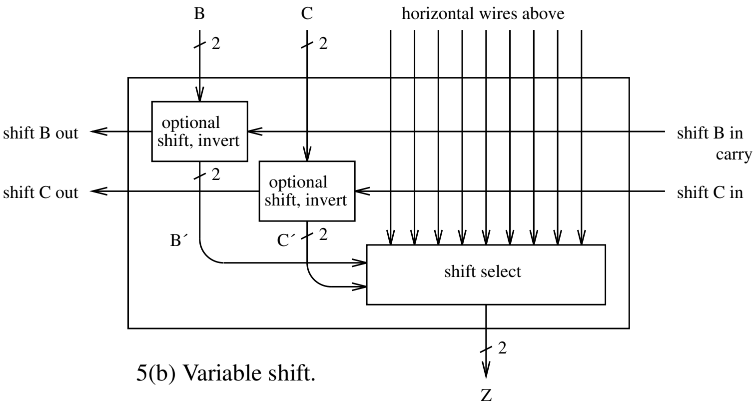


Figure 4: Simplified logic block schematic.

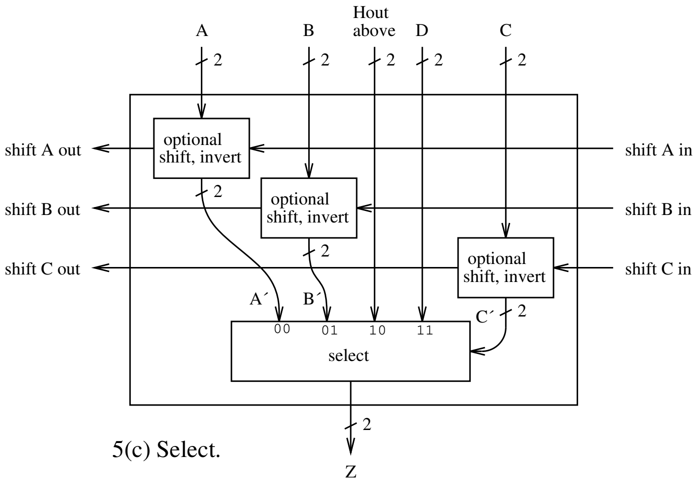
Figure 5: The principal logic block functions. All lookup tables are simultaneously indexed twice to get two single-bit results. (See text.)



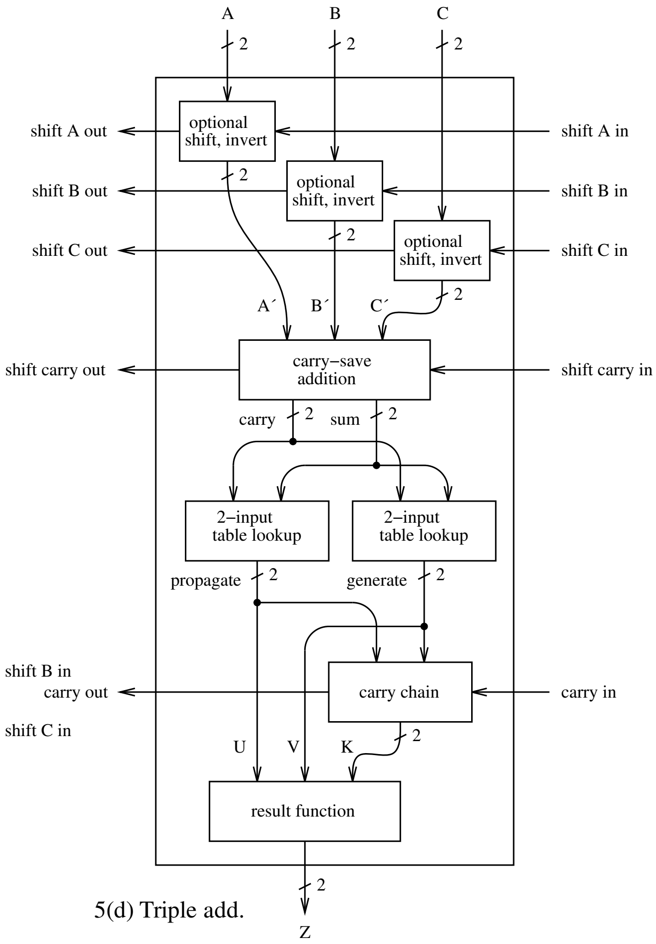
5(a) Table lookup.



5(b) Variable shift.



5(c) Select.



5(d) Triple add.

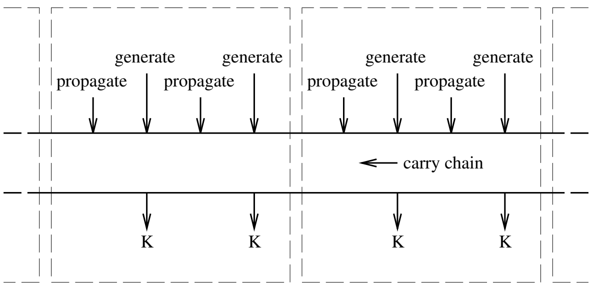


Figure 6: Carry chain box spread across each row. Compare with Fig. 5(d).

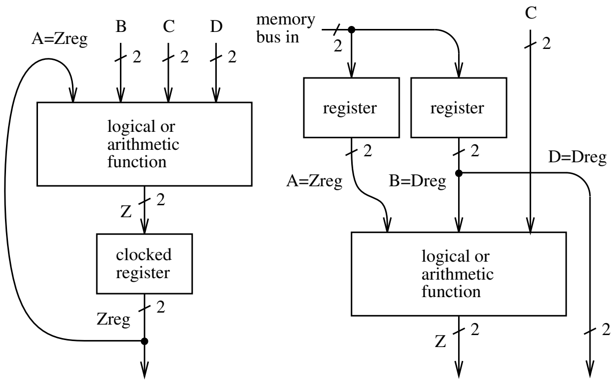


Figure 7: Any of a logic block's A, B, C, or D inputs can be taken from its internal registers.

2.1 Array logic blocks

Each logic block in the array can implement a function of up to four 2-bit inputs. (Recall that the basic data quantum is 2 bits.) Operations on data wider than 2 bits can be formed by adjoining logic blocks along a row (Fig. 3). Construction of multi-bit adders, shifters, and other major functions is aided by hardware invoked through special logic block modes.

Fig. 4 shows the main data paths through a logic block. Four 2-bit inputs (A, B, C, D) are taken from adjacent wires and are used to derive two outputs. One output is calculated (Z), and the other is a direct copy of an input (D). Each output value can be optionally buffered in a register, after which the two 2-bit outputs can be driven onto as many as three pairs of wires leading to other logic blocks. The logic block registers can also be read or written over the memory buses. The "D path" in a logic block allows for a wire-to-wire connection independent of the function calculated by the logic block.

The principal logic block functions (modes) are illustrated in Fig. 5. The four modes in the figure implement:

(a) an arbitrary 4-input bitwise logical function;
(b) a variable shift of up to 15 bits;
(c) a 4-way select (multiplexor) function; and
(d) a 3-input add/subtract/comparison operation.

Each lookup table in Fig. 5 performs the exact same function separately on the high and low bits of its operands. For example, in *table lookup* mode (Fig. 5(a)), a 16-bit lookup table specifies an arbitrary 4-input logical function f. This function is independently applied to the high and low bits of the inputs A', B', C', and D' to generate the high and low bits of the result; that is, $Z_1 = f(A'_1, B'_1, C'_1, D'_1)$ and $Z_0 = f(A'_0, B'_0, C'_0, D'_0)$. The effect is to perform an arbitrary logical function bitwise on the four 2-bit wide inputs.

The decision to make everything 2 bits wide is based on the assumption that a large fraction of most configurations will be taken up by multi-bit operations that are configured identically for each bit. By doubling up bits, the size of configurations—and thus the time required to load configurations and the space taken up on the die to store them—is reduced at the cost of some loss of flexibility.

To support fast 32-bit wide additions, each row includes a fast carry chain "box" spread across all the logic blocks on a row, as shown in Fig. 6. The carry chain is fast enough to be able to perform a full-sized addition in one array clock cycle.

Compared to typical FPGAs, Garp expends more hardware on accelerating operations like adds and variable shifts. In fact, each row of Garp's array approximates a conventional ALU! However, with most of the array die area typically going to inter-block wiring and configuration storage, the incremental area cost of including this special hardware is not necessarily as high as one might think. The cost can be paid back when a configuration that uses the special modes is faster and/or needs fewer logic blocks as a result.

In addition to an adjacent wire, each of the A, B, C, and D inputs has the option of connecting to the Z or D register inside the logic block. This feature enables a number of useful paths within a logic block; Fig. 7 illustrates two examples.

2.2 Array wires

Vertical and horizontal wires exist within the array for moving data between logic blocks. All array wires are grouped into pairs to carry 2-bit quantities. Each pair of wires can be driven by only a single logic block but can be read simultaneously by all the logic blocks spanned by the wires. The wire network is passive, in that a value cannot jump from one wire to another without passing through a logic block.

Fig. 8 illustrates the pattern of vertical wires in a single column of 32 rows, while Fig. 9 shows the horizontal wires between two rows. The horizontal and vertical wires have different patterns because they are optimized for different purposes. The shorter horizontal wires are tailored to multi-bit shifts across a row, while the vertical wires are oriented towards connecting functional units laid out horizontally. The long horizontal wires are typically used to broadcast control signals to the logic blocks of a single multi-bit operation.

The driver of every wire is fixed by a configuration and cannot be changed without loading a new configuration. Configurations are checked by the hardware when loaded to ensure that no wire has more than one driver. Configurations failing this test cannot be loaded.

2.3 Array timing

Commercial FPGAs usually specify precise delay times for all array components. It is a development task (either for the tools or for a human designer) to ensure that no signal path exceeds its maximum allowed delay. In practice, the relationships between different components' delay times will vary with each FPGA implementation. This makes it harder to

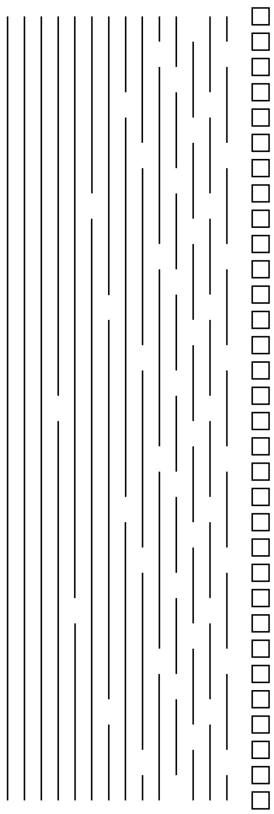


Figure 8: The pattern of vertical wires (V wires) in a single column of 32 rows. Each line drawn actually represents a pair of wires (2 bits).

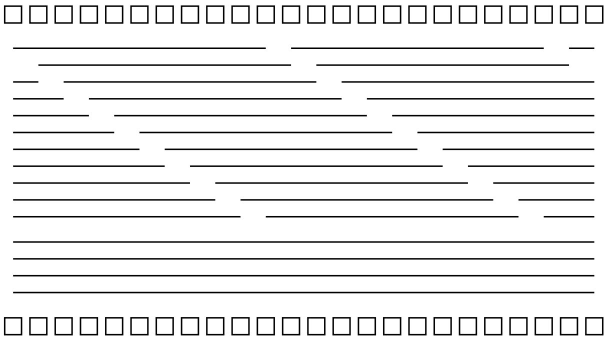


Figure 9: The horizontal wires (H wires and G wires) between two rows. Again, each line represents a pair of wires (2 bits).

Multiplier	Rows	Cycles
32 bits $\times$ 5-bit constant $\rightarrow$ 32 bits	1	1
32 bits $\times$ 8-bit constant $\rightarrow$ 32 bits	2	2
32 bits $\times$ 16-bit constant $\rightarrow$ 32 bits	4	3
16 bits $\times$ 16 bits $\rightarrow$ 32 bits	4	7
16 bits $\times$ 16 bits $\rightarrow$ 32 bits	9	5

Figure 10: Statistics on multipliers synthesized in the array.

predict the speeds at which two versions from the same FPGA family will execute the same configuration.

Rather than specify component delays as precise times that would change with each processor generation, delays in Garp are defined in terms of the sequences that can be fit within each array clock cycle. Only three sequences are permitted:

– short wire, simple function, short wire, simple function;
– long wire, any function not using the carry chain; or
– short wire, any function.

Any other sequence must be assumed to require multiple clock cycles. The *short* wires include all the shorter horizontal wires, plus vertical wires less than a certain length. A *simple* function is either a direct table lookup (Fig. 5(a)) or a traversal of the independent "D path" in a logic block (Fig. 4). At the end of a cycle, a computed value may be latched in a logic block register without affecting the timing.

2.4 Multipliers

Like most FPGAs, multipliers in the reconfigurable array must be built up out of smaller parts, typically selectors and adders. Nevertheless the importance of multiplication has had its impact on the design of the Garp array. The ability to add three operands instead of only two (*triple add* mode, Fig. 5(d)) exists as much to support fast multiplication as for any other purpose. A special variation on *select* mode (Fig. 5(c)) also makes it easier to choose from among partial products 0, A, $2A$, and $3A$.

Fig. 10 lists the area and time delays of various multipliers synthesized in the array. Multiplies by constants are especially dense because they can be configured as hard-wired shifts and adds using the horizontal wires between rows and *triple add* mode.

2.5 Processor control of array execution

The main processor has a number of instructions for controlling the array. The most important are listed in the table of Fig. 11. These include instructions for loading configurations, for copying data between the array and the processor registers, for manipulating the array clock counter, and for saving and restoring array state on context switches.

As mentioned earlier, a *clock counter* controls array execution. While the clock counter is zero, the array is stopped. By setting the clock counter positive, the main processor can make the array execute for a specific number of clock cycles. The clock counter decrements with each array clock cycle until reaching zero.

To avoid restricting the main processor implementation, the Garp Architecture does not specify how many main processor instructions might execute during each array clock cycle. Instead, to keep processor and array synchronized, many of the new processor instructions (Fig. 11) first wait for the array clock counter to reach zero before performing their function. The simplest example is when the main processor needs to read the result of a computation performed by the array. After setting the array clock counter to the proper value, the processor can execute a `mfga` instruction at any time. If the array is not yet done, `mfga` will wait for the array clock counter to become zero before attempting to copy the result over to the processor.

Instruction	Interlock?	Description
`gaconf` *reg*	yes	Load (or switch to) configuration at address given by *reg*.
`mtga` *reg*, *array-row-reg*, *count*	yes	Copy *reg* value to *array-row-reg* and set array clock counter to *count*.
`mfga` *reg*, *array-row-reg*, *count*	yes	Copy *array-row-reg* value to *reg* and set array clock counter to *count*.
`gabump` *reg*	no	Increase array clock counter by value in *reg*.
`gastop` *reg*	no	Copy array clock counter to *reg* and stop array by zeroing clock counter.
`gacinv` *reg*	no	Invalidate cache copy of configuration at address given by *reg*.
`cfga` *reg*, *array-control-reg*	no	Copy value of array control register *array-control-reg* to *reg*.
`gasave` *reg*	yes	Save all array data state to memory at address given by *reg*.
`garestore` *reg*	yes	Restore previously saved data state from memory at address given by *reg*.

Figure 11: Basic processor instructions for controlling the reconfigurable array. The *Interlock?* column indicates whether the instruction first stalls waiting for the array clock counter to run down to zero. (Instructions can be interrupted while stalled.) The last three instructions are intended for context switches.

The `mtga` and `mfga` instructions copy to and from the middle 16 logic blocks of a row. (Recall Fig. 2.) Additional instructions (not shown) give the processor access to the logic blocks at the edges of the array, and also make it possible to send the values of two registers to the array in one step.

2.6 Configurations

Each block in the array requires exactly 64 configuration bits (8 bytes) to specify the sources of inputs, the function of the block, and any wires driven with outputs. No configuration bits are needed for the array wires, so a configuration of 32 rows requires exactly $8 \times 24 \times 32 = 6144$ bytes. Assuming a 128-bit path to external memory, loading a full 32-row configuration takes 384 sequential memory accesses. A typical processor external bus might need 50 μs to complete the load.

Since not all useful configurations will require the entire resources of the array, Garp allows partial array configurations. The smallest configuration is one row, and every configuration must fill exactly some number of contiguous rows. When a configuration is loaded that uses less than the entire array, the rows that are unused are automatically made inactive.

Distributed within the array is a cache of recently used configurations, similar to an ordinary instruction cache. The size of this cache is implementation-dependent. A reasonable Garp might have a 4-deep cache at every logic block—sufficient to hold four 32-row configurations, or sixteen 8-row configurations, or any other combination of the same size.

Note that in order to maximize cache utilization, partial configurations are not necessarily loaded at the first physical row of the array. The hardware translates row numbers so that programs see all configurations as starting at logical row 0. Exactly where partial configurations can be placed in the array is dependent on the pattern of vertical wires (Fig. 8). The vertical wires in Garp follow a repeated, recursive pattern so that partial configurations can be loaded at various offsets.

Two configurations can never be active at the same time, no matter how many array rows might be left unused by a small configuration. This is analogous to there being only one thread of control—only one program counter—in the main processor. If two independently-written configurations could be active simultaneously, there is no way to guarantee they would not interfere with each other's use of the vertical wires. If a program has a special need for making more than one configuration active at a time, it can easily load one larger configuration containing both the smaller ones.

2.7 Array access to memory

Memory accesses can be initiated in the array without direct processor intervention. These memory accesses proceed in two phases: the first phase is the memory access *request*, and the second is the *data transfer*. For writes, the two phases may occur on the same array cycle. For reads, the memory request necessarily precedes the data transfer. With some restrictions, the two phases can be pipelined so that a new memory access can be initiated every cycle.

Array memory accesses are controlled by the control blocks at the edge of the array. Parallel to the memory buses (Fig. 2), an *address bus* also runs vertically through the rows. Control signals requesting memory accesses can be generated in the array logic blocks and forwarded by the control blocks to the memory system. A memory address is then read over the address bus from the Z registers of a selected row. The data is transferred over a memory bus to/from another selected row, which is usually a different row than the one that supplied the address. Up to four contiguous 32-bit words can be read or written with one request over the four memory buses.

The array sees the same memory hierarchy as the main processor. Misses in the on-chip data cache cause array execution to be stalled while the data is fetched from external memory. To reduce cache misses, the array can perform prefetching accesses that merely load the on-chip data cache. Page faults due to array memory accesses are also possible and cause the faulting process to be suspended while the page fault is serviced.

In contrast to many commercial FPGAs, Garp's array contains only a modicum of writable storage—4 bits per logic block. This was a conscious design decision intended primarily to limit the time needed for context switches of running processes. The existing on-chip data cache provides ample temporary storage; although the limited bandwidth of the memory buses can in theory be a bottleneck.

In addition to the mechanism for demand accesses just described, the array also has available to it three memory queues for performing read-aheads and write-behinds on multiple data streams. At least two input streams and one output stream are supported. All three streams can be read/written in the same cycle, using three of the memory buses concurrently. Memory queues are programmed by the main processor before a configuration is executed.

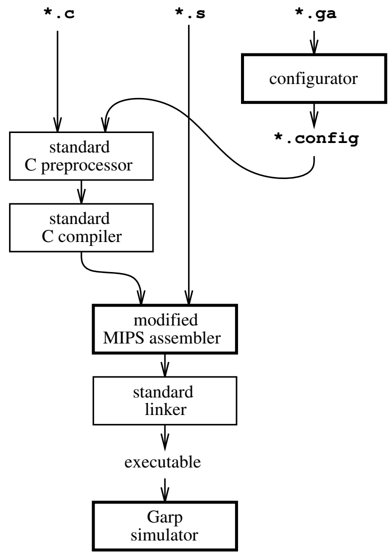


Figure 12: The Garp programming environment. New or modified tools are highlighted.

3 Software Environment

Software tools have been created that make it possible to write programs for Garp and then simulate them with approximate clock-cycle accuracy. The software path is summarized in Fig. 12. Only two tools are substantially new: the *configurator* and the Garp simulator. The Garp assembler is merely a modified MIPS assembler.

An array configuration is coded in a "`.ga`" file in a simple textual language. This source is fed through a program called the *configurator* to generate a representation of the configuration as a collection of bits. For simplicity, the configurator creates a text file that can be used as a character array initializer in a C program.

The only need for assembly language programming is to invoke the Garp instructions that interface with the reconfigurable array. Since we are using the Gnu C Compiler (`gcc`), the same could be accomplished with inline '`asm`' statements.

3.1 The configurator

The configurator accepts a human-readable description of a configuration and converts it to the binary representation used by the processor. The input language to the configurator is more akin to an assembly language than to either a high-level language or the typical FPGA netlist. Data and operations must be placed explicitly within rows and columns by the programmer. A configuration is defined as a collection of rows, with each row containing within it logic blocks in specific columns. The basic syntax is

```
row optional-row-name:
{
  column-number(s): logic-block-settings;
  ...
}
...
```

A feel for the permissible *logic-block-settings* is probably easiest to impart by example. The following specifies a complete configuration for adding three 32-bit values in columns 4–19 (the middle 16 columns of the array):

```
row .a: --Row 0
{
--Drive Z registers onto V wires.
 4-19:  A(Zreg),function(A),Vout(Z);
--Drive D registers onto H wires below.
 4-19:  D(Dreg),Hout(D);
}

row :   --Row 1
{
--Add D registers + values from row 0;
--latch result in Z registers.
 4:     shiftzeroin;
 4-19:  A(.a),B(above),C(Dreg),add3,
         U(carry^sum),V(sum),
         result(U^K),bufferZ;
}
```

In this configuration, the values in the Z and D registers of row 0 and in the D registers of row 1 are added together and their sum stored in the Z registers of row 1. Column 4 is the least significant (rightmost) of the 16 columns. Row names (e.g., `.a`) must begin with a period to distinguish them syntactically.

The '`A(.a)`' field in the second row specifies that the A input for those logic blocks is to come from the row labeled `.a`—in this case, the first row. To obtain a connection through vertical wires, the programmer merely names the source needed for a logic block input. It is the responsibility of the configurator to choose specific vertical wires for making the connections. The A inputs in row 1 of the example are thus taken over vertical wires from row 0. The rather different syntax '`B(above)`', on the other hand, indicates that the B inputs are to be read from row 0 over the *horizontal* wires between the two rows.

For the example given, the output from the configurator is the text

```
{ 0x00, 0x00, 0x00, 0x02, ... 0x00, 0x00 }
```

which is suitable for initializing a C `char` array. (Of course, the majority of the output has been elided here.)

3.2 Linking a configuration into a C program

The reconfigurable array is only used within the time consuming parts of a program where it can be usefully employed. The remainder of the program is written in C, is compiled with an ordinary C compiler, and is executed on the main processor without reference to the reconfigurable array. A configuration thus has to be linked into an ordinary C program.

Continuing with the example above, if the configurator output is in a file called '`add3.config`', the C code

```
char config_add3[] =
#include "add3.config"
;
```

suffices to initialize a C array `config_add3` with the desired configuration bits. This makes the configuration accessible to the program; however, it will still have to be loaded and activated in the array to actually do something. Since a configuration can only be invoked with the new Garp-specific instructions that are unknown to the compiler, some assembly language programming is required.

The following assembly code loads and executes the same example (refer back to Fig. 11):

```
add3: la v0,config_add3
              # v0 now contains pointer
              #  to config_add3 array.
      gaconf v0
      mtga a0,$z0
      mtga a1,$d0
      mtga a2,$d1,2
              # Step array 2 cycles.
      mfga v0,$z1
      j ra    # Return from subroutine.
```

The names `v0`, `a0`, `a1`, `a2`, and `ra` refer to ordinary MIPS registers; `la` is the MIPS "load address" instruction. The symbols `$z0` and `$d0` indicate the Z and D registers of array row 0; `$z1` and `$d1` are the same for row 1. The MIPS subroutine calling convention passes the first three subroutine arguments in registers `a0`, `a1`, and `a2`, with the subroutine return value being passed back in register `v0`.

With this assembly language stub, a program can add any three values a, b, and c using the reconfigurable array by executing the ordinary subroutine call `add3(`a`,`b`,`c`)`. The `add3` subroutine first loads the proper configuration into the array (or switches to it, if it is already in the array's configuration cache). It then copies its three arguments into array registers, steps the array for 2 cycles to perform the addition, reads the sum back into `v0`, and returns.

Of course this example involves too much overhead. In practice, the array would be used for something substantial that could not just as easily be done in the main processor.

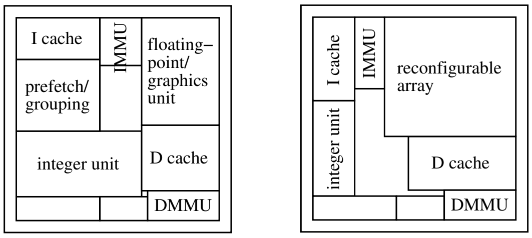
3.3 The simulator

A complete hardware implementation of Garp does not yet exist, so Garp programs must be executed on a simulator. The simulator loads and executes standard MIPS executables. Operating system calls are forwarded to the environment in which the simulator is running.

Outside of operating system calls, the simulator does its best to count true clock cycles. Interlocks that stall instructions are noticed and stall cycles counted. Memory caches are also modeled, so that cache miss stalls can be added in. The simulator assumes the main processor is only a simple single-issue MIPS. Although the simulator is unlikely to be cycle-for-cycle identical with any actual implementation, its cycle counts should be realistic.

4 Preliminary Evaluation

In an attempt to evaluate the Garp Architecture, we compared a hypothetical Garp against a Sun UltraSPARC 1/170. The UltraSPARC is a 4-way superscalar 64-bit processor with 16 kB each of on-chip instruction and data caches. The processor runs at 167 MHz, and is implemented in a 0.5 μm, 4-metal-layer process in a die size of 17.5×17.8 mm^2.



(a) UltraSPARC. (b) Hypothetical Garp.

Figure 13: Floorplan of the UltraSPARC die, and that of a hypothetical Garp die constructed in the same technology.

Benchmark	167 MHz SPARC	133 MHz Garp	ratio
DES encrypt of 1 MB	3.60 s	0.15 s	24
Dither of 640×480 image	160 ms	17 ms	9.4
Sort of 1 million records	1.44 s	0.67 s	2.1

Figure 14: Benchmark results. The times for Garp are obtained from program simulation.

To create the hypothetical Garp, we removed the SPARC's superscalar integer and floating-point processing units from the UltraSPARC die and replaced them with a MIPS processor extended with Garp's reconfigurable array. The new processor is a single-issue 32-bit MIPS-II, which is rather smaller and less powerful than the UltraSPARC's processing unit. Fig. 13 shows die floorplans of the actual UltraSPARC and the hypothetical Garp derived from it. This surgery essentially puts a Garp on top of an UltraSPARC memory system. The Garp simulator was of course retargeted to model the SPARC caches as closely as possible.

The size assumed for the reconfigurable array (Fig. 13(b)) draws on our experience with a tentative VLSI array implementation. We also assumed conservatively that we could only achieve 80% of the SPARC clock rate for Garp's array clock. This restricts Garp's clock to 133 MHz versus the SPARC's 167 MHz.

The time taken by the two processors on three benchmarks are tabulated in Fig. 14. Summaries of the benchmarks are given in the next sections. The results shown are in each case the best we were able to attain for that processor. Given the approximations involved, the Garp numbers unfortunately must be considered rough; however, it is easy to see that Garp holds an advantage for at least some problems.

4.1 Data Encryption Standard (DES)

One of the most important encryption methods over the last 20 years has been the Data Encryption Standard, or DES [12]. DES is a good application for reconfigurable hardware because normal processors have trouble implementing it efficiently.

DES encrypts 64 bits of data at a time. Each 64 bits is run through an "obfuscation loop" 16 times; and it is in this loop that DES spends most of its time. The 64 bits are first divided into two 32-bit quantities R_{-1} and R_0, and then the following steps are repeated for $i = 1$ up to 16 (see Fig. 15):

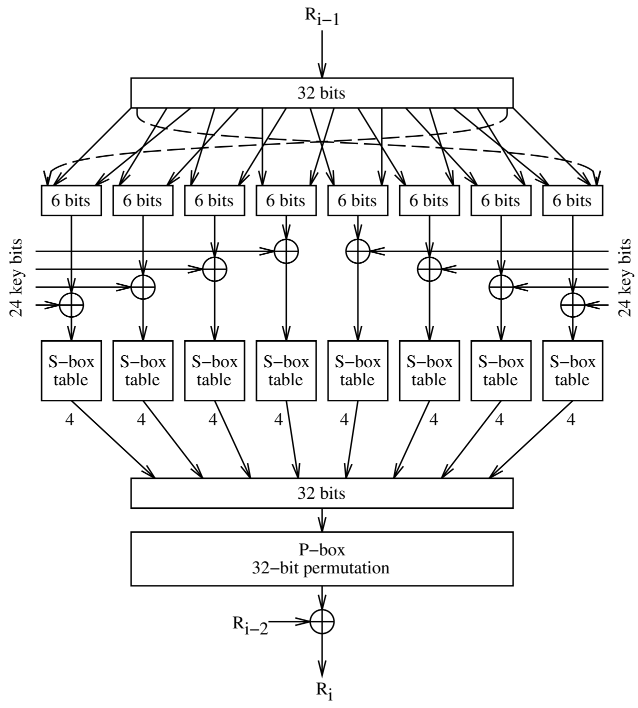


Figure 15: One iteration of the inner loop of DES. The $\oplus$ symbols indicate exclusive-or operations.

1. Extract eight 6-bit subsequences from R_{i-1}, and exclusive-or these with 48 bits from the encryption key.
2. Apply each of the resulting 6-bit values as an index into an "S-box" table of 4-bit values. (Each S-box is unique and approximates a random function.)
3. Perform a permutation on the 32 bits of S-box results. (This permutation is always the same.)
4. Exclusive-or the permuted result with the older R_{i-2} to form the new R_i.

At the end, the encrypted 64-bit output is formed from R_{15} and R_{16}.

Software implementations of DES invariably implement the S-boxes as table lookups requiring a read from memory for each S-box evaluation. All told, $16 \times 8 = 128$ table-lookup memory reads are needed for each 64 bits encrypted. On the other hand, good software implementations can avoid the final 32-bit permutation by pre-permuting the S-box table entries. This makes the table entries a full 32 bits in size, but the eight S-box outputs need only be or-ed together before being combined with R_{i-2}.

Unlike software, any sufficiently large FPGA can implement this algorithm directly. The S-box table lookups and all the bit permutations can be done quickly and in parallel, without reference to external memory. Garp needs only 6 cycles per inner loop iteration. Our simulations indicate that a 133 MHz Garp could be 24 times faster than the 167 MHz UltraSPARC for this task.

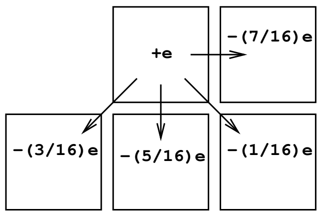


Figure 16: Floyd-Steinberg error diffusion. An image is dithered from top to bottom in scan order. Replacing a pixel's original color with the closest available color results in a color error e. This error gets pushed to 4 as-yet-uncommitted neighboring pixels by adjusting the original colors at those pixels. The process repeats with the next pixel to the right.

4.2 Image dithering

As another benchmark, we considered the dithering of a full-color 640×480 image to a fixed palette of fewer than 256 colors. The input image stores 3 bytes per pixel, for a total of 256 levels each of red, green, and blue for each pixel. The target palette in our benchmark is the so-called "Web palette" used by Web browsers such as Netscape Navigator. This palette contains $216 = 6^3$ colors in an orthogonal arrangement, with 6 levels each of red, green, and blue. To dither to this palette we employed Floyd-Steinberg error diffusion [13], which is essentially the standard algorithm for this task.

The dithering of an image proceeds from top to bottom in scan-line order. Dithering each pixel involves the following two steps:

1. Find the color in the target palette closest to the given pixel color.
2. Find the color error introduced by using a not-quite-correct color, and distribute this error to neighboring pixels by adjusting the neighbors' colors.

Fig. 16 shows how a pixel's color error is distributed (diffused) to its neighbors in the Floyd-Steinberg algorithm.

In our case, finding the closest target color is a matter of reducing the source image's 256 levels each of red, green, and blue to the 6 levels each in the target palette. This is accomplished by dividing each color component by 51 and rounding. Calculating the error requires multiplying the result back by 51 and subtracting. Distributing the error involves four scales and additions to neighboring pixels. To save some work, errors diffused to a single pixel by multiple of its neighbors are added together before being added into the destination pixel.

For this application, Garp is found to be over 9 times faster than the UltraSPARC. Garp's advantage comes from its ability to manipulate 8-bit quantities more adeptly. On both Garp and the UltraSPARC, the division by 51 is done by multiplying by an approximation to $1/51$. Multiplies are implemented on both in terms of shifts and adds, which Garp can do fairly efficiently.

4.3 Sorting

The third benchmark we examined is the sorting of an array of one million (actually 2^{20}) records, where each record is a key, value pair. The sort orders the records according to their 32-bit keys. The corresponding 32-bit values are not interpreted but must be correctly permuted with the keys.

On the SPARC, the fastest sort found was a pure quicksort, taking 1.44 s. Estimates are that close to half that time goes to memory accesses. For N records, quicksort in general makes at least $\log_2 N$ partitioning passes over the entire array. Given in our case $N = 2^{20}$, the number of partitioning passes is clearly about 20.

For Garp, we reduced memory traffic by doing a merge sort using merges of 9 streams at a time. Each merge takes 9 sorted streams as input and outputs a single sorted stream nine times as long. This cuts the number of passes needed to $\log_9(2^{20})$, or about 7. (The odd number 9 helps reduce cache conflicts.) Garp at 133 MHz should be able to perform the complete sort in 0.67 s, roughly twice as fast as the 167 MHz SPARC.

All our efforts to apply the same sophistication on the SPARC have not done as well as quicksort on that machine. More time is always introduced in additional instructions than is ultimately saved.

5 Ongoing Work

The results in the previous section must be taken as preliminary because the Garp implementation is as yet unproved. Our first priority now is to better resolve Garp's performance by completing a VLSI implementation of Garp's reconfigurable array.

We have also just scratched the surface of potential applications. Likely areas include cryptography, compression, pattern matching, signal processing, and graphics. On a different tack, another researcher in our group, Tim Callahan, is examining the potential performance gains from having a compiler automatically map important program loops to a Garp-like array.

Finally, we would like to make some comparisons against other possible uses for the same silicon area, such as a vector unit. Reconfigurable hardware is unlikely to be the best solution for all problems, so it would be good to characterize its limitations along with its advantages.

6 Conclusions

Because Garp is an extension of existing computing practice, a Garp-like architecture has a better chance of becoming part of the mainstream than FPGA-only machines. The results so far suggest that a Garp processor would have a substantial advantage over standard RISC processors for some applications. We believe the hypothetical Garp outlined in the paper could easily be built today; even larger arrays should be possible in the near future. If silicon densities continue to grow as they have historically, reconfigurable hardware in some form may well become an inevitable component of future processors.

Acknowledgments

We would like to specially thank Krste Asanović, Tim Callahan, William Chang, and André DeHon for their assistance and comments concerning this paper, and for many conversations on the topics contained therein.

References

[1] Peter M. Athanas and Harvey F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, pp. 11–18, Mar. 1993.

[2] André DeHon, "DPGA-coupled microprocessors: Commodity ICs for the early 21st century," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1994, pp. 31–39.

[3] André DeHon, *Reconfigurable Architectures for General-Purpose Computing*, Ph.D. thesis, Massachusetts Institute of Technology, 1996.

[4] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, and Douglas Sweely, "Building and using a highly parallel programmable logic array," *Computer*, vol. 24, no. 1, pp. 81–89, Jan. 1991.

[5] Eric Lemoine and David Merceron, "Run time reconfiguration of FPGA for scanning genomic databases," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1995, pp. 90–98.

[6] Bernard Gunther, George Milne, and Lakshmi Narasimhan, "Assessing document relevance with runtime reconfigurable machines," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1996, pp. 10–17.

[7] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé H. Touati, and Philippe Boucard, "Programmable active memories: Reconfigurable systems come of age," *IEEE Transactions on VLSI*, vol. 4, no. 1, Mar. 1996.

[8] Rahul Razdan and Michael D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, Nov. 1994, pp. 172–180.

[9] Ralph D. Wittig and Paul Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1996, pp. 126–135.

[10] Xilinx, *The Programmable Logic Data Book*, 1994.

[11] Michael J. Wirthlin and Brad L. Hutchings, "A dynamic instruction set computer," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1995, pp. 99–107.

[12] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd edition, John Wiley and Sons, 1996.

[13] Robert Ulichney, *Digital Halftoning*, MIT Press, Cambridge, Massachusetts, 1987.