The Garp Architecture

John R. Hauser

University of California at Berkeley
Department of Electrical Engineering and Computer Sciences
Computer Science Division

October 1997

This work was supported in various parts by DARPA grant DABT63-96-C-0048, ONR grant N00014-92-J-1617, NSF TITAN grant CDA 94-01156, and the California State MICRO Program.

Contents

1 Introduction

The Garp processor architecture combines an industry-standard MIPS processor with a new *reconfigurable* computing device that can be used to accelerate certain computations. Figure 1 shows the organization of this architecture at the highest level. The core of Garp is an ordinary processor supporting the MIPS-II instruction set. Added to this is something called a *reconfigurable array*, which is a two-dimensional array of small computing elements interconnected by a network of wires. Garp's reconfigurable array somewhat resembles *field-programmable gate arrays* (*FPGAs*) available from Xilinx, Altera, and other manufacturers.

Each computing element in the reconfigurable array can perform a simple logical or arithmetic operation on operands that are at most 2 bits in size. Larger computations are achieved by aggregating these array elements into larger computational circuits. The function of each array element and the connections between the elements are determined by a *configuration* of the array, which is loaded under the direction of the main processor. The array's configuration can be changed as often as desired, allowing the array to be applied to various pieces of a computation over time.

Use of the reconfigurable array is controlled exclusively by the program executing on the main processor. Although a program can execute entirely on the main processor without referencing the reconfigurable array at all, certain computations can be completed faster by the array than by the main processor. Thus it is expected that for certain loops and/or subroutines, programs will switch execution temporarily to the array to obtain a speedup.

This document defines the Garp architecture by detailing Garp's extensions to the MIPS-II architecture. Documentation for the MIPS-II architecture can be found elsewhere.

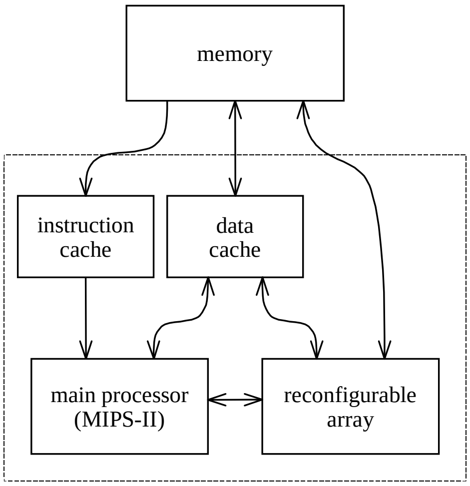


Figure 1: Basic organization of Garp.

The reconfigurable array itself is described first in Section 2, after which Section 3 covers the integration of the array with the main processor and memory system.

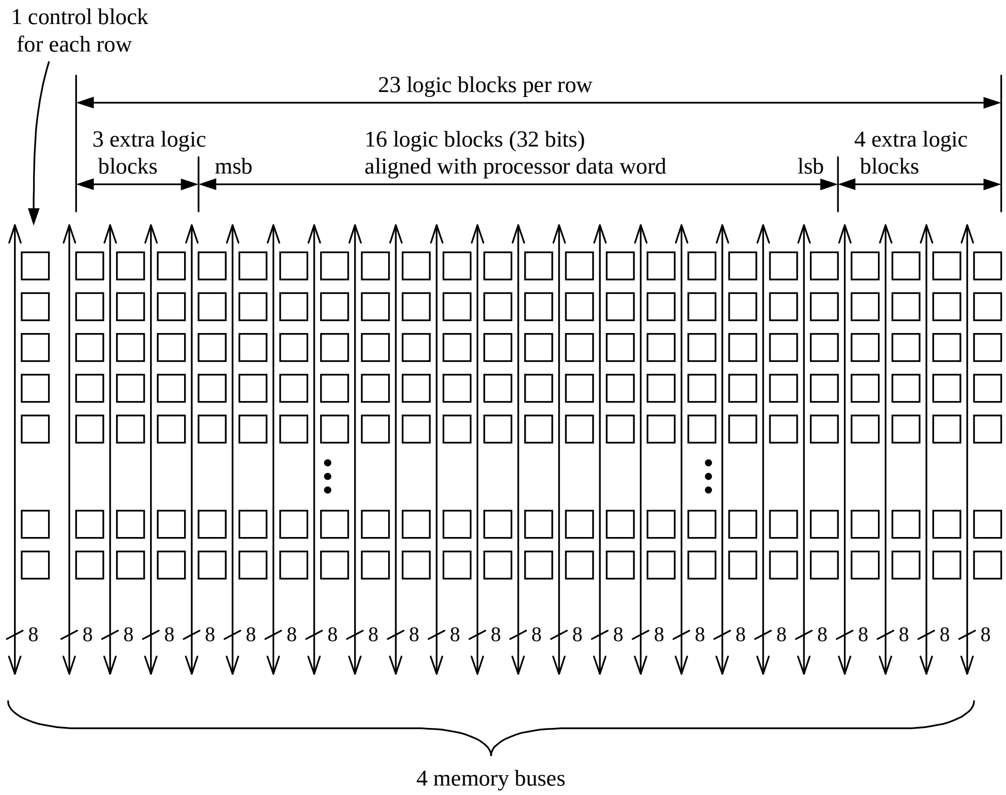


Figure 2: Organization of the reconfigurable array. In addition to the memory buses, the array blocks are connected by an internal wire network (not shown).

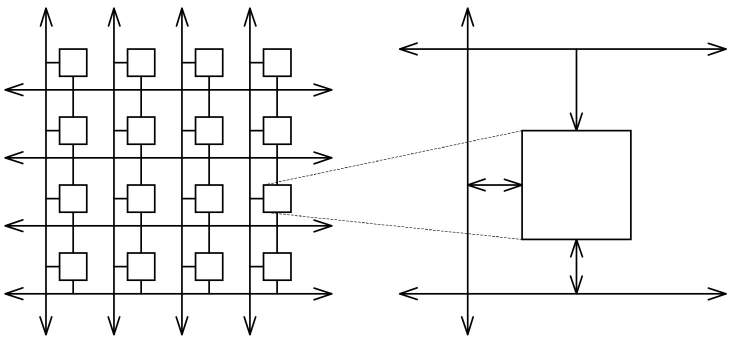


Figure 3: Internal wiring within the array (independent of the memory buses). Here each arrow represents multiple physical wire paths.

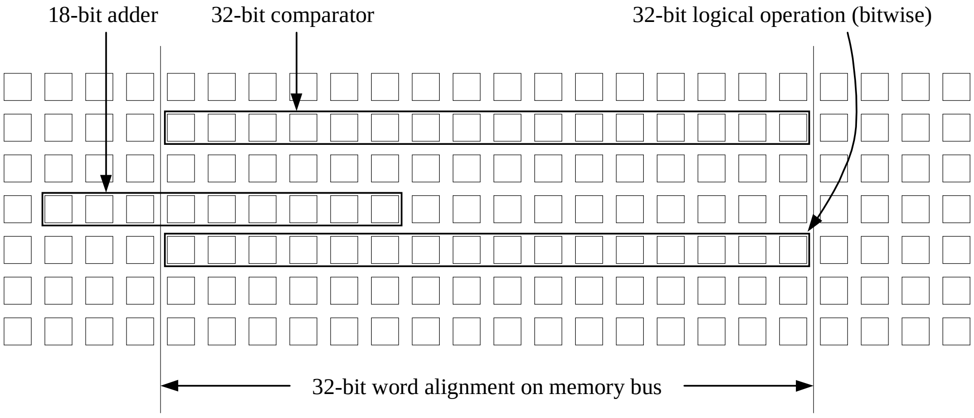


Figure 4: Typical natural layouts of multi-bit functions.

2 Reconfigurable array

The core of the reconfigurable array is a two-dimensional matrix of small processing elements called *blocks* (Figure 2). One block on each row is known as a *control block*, and the rest of the blocks are *logic blocks*. The number of columns of blocks is fixed at 24. The number of rows is implementation-specific, but can be expected to be at least 32.

The basic "quantum" of data within the array is 2 bits. All wires are organized in pairs to transmit 2-bit quantities, and logic blocks operate on these values as 2-bit units. Operations on 32-bit quantities thus generally require 16 logic blocks.

As Figure 1 shows, the array has access to the standard memory hierarchy of the main processor. Four *memory buses* run vertically through the rows for moving information into and out of the array (Figure 2). During array execution, the memory buses are used for moving data to and from memory and/or the main processor. For memory accesses, transfers are limited 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 bandwidth of the memory buses is used.

The memory buses are not available for moving data between array blocks. An internal *wire network* provides connections between blocks. Wires of various lengths run orthogonally vertically and horizontally. Figure 3 summarizes the available wire paths. Vertical wires can be used to communicate between blocks in the same column, while horizontal wires can connect a block to others in the same row or in the next row below. 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.

In addition to performing a minor computation, each logic block can hold a few bits of data in registers. These data registers are latched synchronously according to an *array clock*, the frequency of which 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. As is true for the main

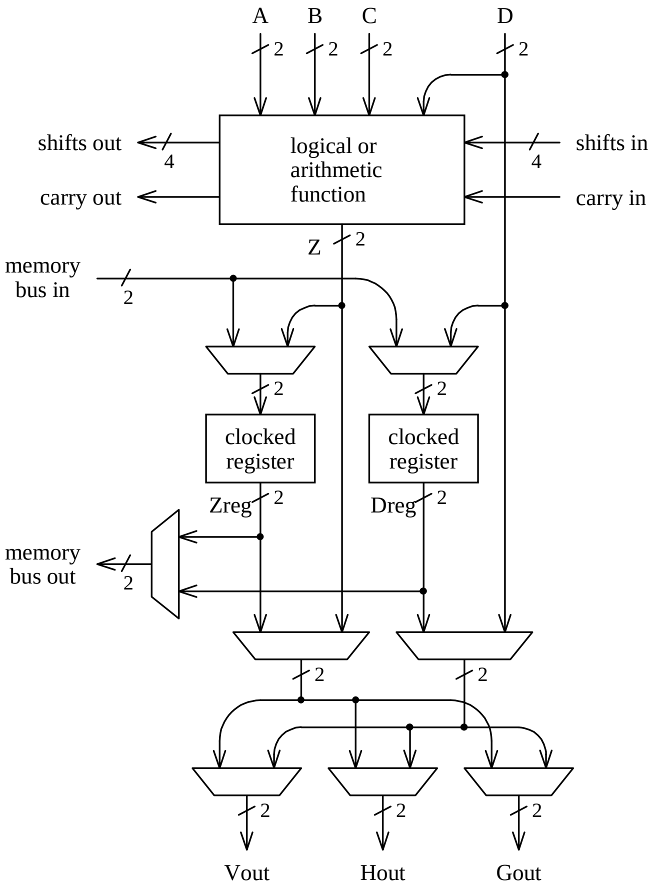


Figure 5: Simplified logic block schematic.

processor, the array clock governs the progress of a computation in the array.

Each logic block can implement a function of up to four 2-bit inputs. Operations on data wider than 2 bits can be formed by adjoining logic blocks along a row (Figure 4). Construction of multi-bit adders, shifters, and other major functions along a row is aided by hardware invoked through special logic block modes. In particular, a fast carry chain runs right-to-left across each row to facilitate large adders and comparators that execute in a single array clock cycle. Since there are 23 logic blocks per row (the leftmost block on each row is a control block), 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, extended data widths, or whatever is needed.

Figure 5 shows the main data paths through a logic block. Four 2-bit inputs (A, B, C, and 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 next few subsections cover the core array architecture in more detail: first the inter-block wire network, then the logic block data paths, and finally the available logic block functions (illustrated in Figure 5 as a non-descript box). Discussion about the control blocks and the memory buses are deferred until the integration of the array with the main processor is covered in Section 3.

2.1 Internal wire network

Internal wires run vertically and horizontally within the array for moving data between logic blocks. All wires in the network 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 logic blocks spanned by the pair. The wire network is passive, in that a value cannot jump from one wire to another without passing through a logic block.

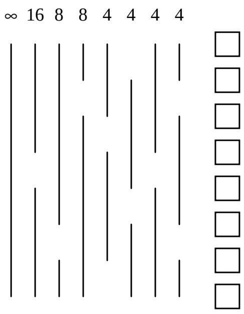
The internal wires are divided into three groups: the *vertical wires* (also called *V wires*), the *global horizontal wires* (*G wires*), and the *local horizontal wires* (*H wires*). Wires running horizontally between logic block rows are either global (G wires) or local (H wires). The G wires span the entire 24-block width of the array, while the H wires normally span exactly 11 blocks. Only the V wires run vertically, but unlike the horizontal wires, they come in a range of lengths.

The pattern of horizontal wires is not the same as that of the vertical wires, so the vertical and horizontal dimensions of the array are not symmetric. This asymmetry is due to the preference for aligning multi-bit operations across rows and not columns. Nevertheless, the columns are all identical amongst themselves; and the rows are also all identical.

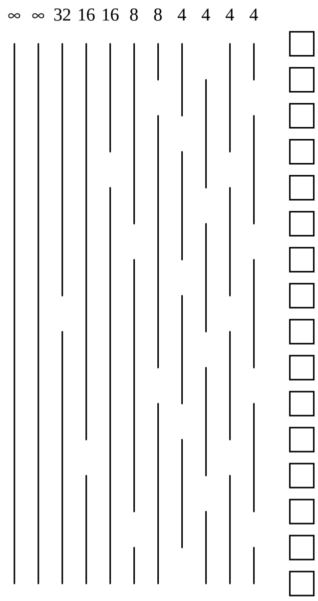
The three categories of wires (V, G, and H) are described in turn below.

2.1.1 Vertical wires (V wires)

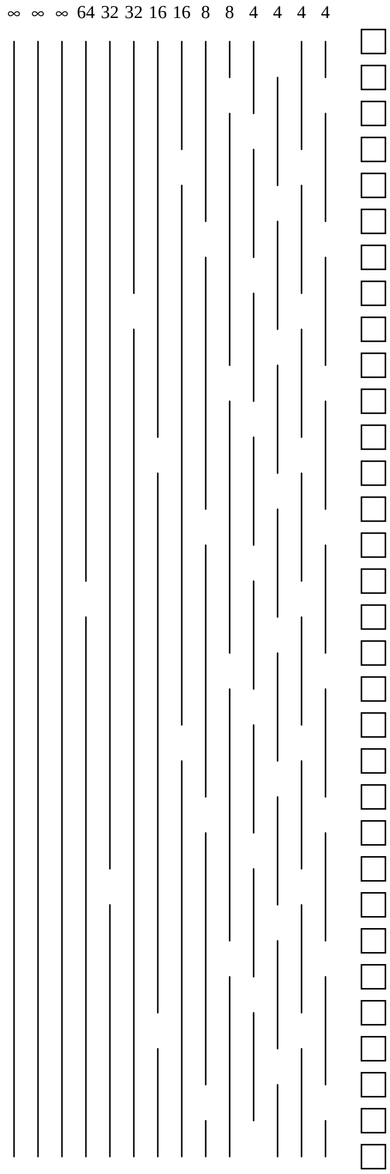
Each column of array blocks has a set of vertical wires (V wires) for making connections among the blocks in that column. Because the number of rows in the array is not strictly fixed, the amount of vertical wiring available depends on the number of rows a configuration has. Figure 6 illustrates the patterns of vertical wires for configurations with 8, 16, and



(a) Vertical wires for 8 rows.

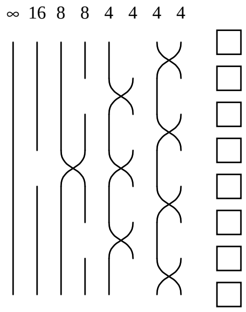


(b) Vertical wires for 16 rows.

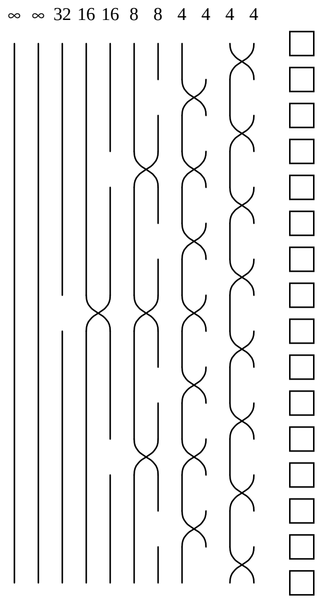


(c) Vertical wires for 32 rows.

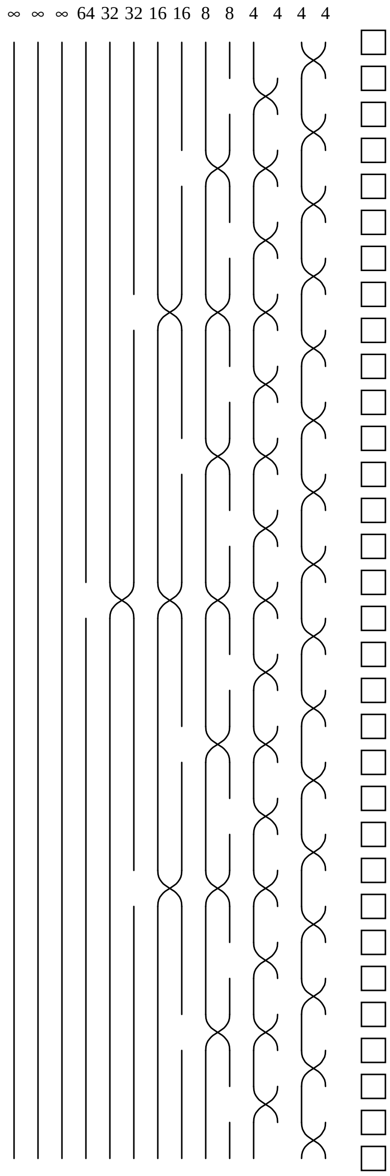
Figure 6: The vertical wires (V wires) for arrays of various sizes. The boxes represent a single column of the array. Each line drawn actually represents a pair of wires (2 bits). Each wire pair can connect to all of the blocks it spans vertically. The numbers at the top give the nominal lengths of different wires.



(a) Vertical wires for 8 rows.



(b) Vertical wires for 16 rows.



(c) Vertical wires for 32 rows.

Figure 7: Twisting of the vertical wires to obtain a recursive structure. Compare with Figure 6. Note that the pattern of vertical wires for an 8-row array is repeated in the upper and lower 8 rows of a 16-row array. The 16-row pattern is in turn repeated in the upper and lower halves of a 32-row array.

32 rows. Each V wire spans a specific set of blocks, any one of which can be configured to drive the wire. All logic blocks spanned by a wire can read from that wire simultaneously. By configuring the vertical wires of several columns in concert, multi-bit values are easily moved among array rows.

Each pair of wires has a nominal length, shown along the tops of the subfigures in Figure 6. Except for some wires with nominally infinite length (global vertical wires), the nominal lengths of wires are all powers of 2. The actual length of a wire can be shorter than its nominal length if the wire would extend above the topmost row or below the bottommost row (or both). Thus although a 32-row array includes wires with nominal length 64, no such wire is longer than 32 blocks in reality. The same obviously applies for the global wires labelled as having infinite length.

Each doubling in the number of rows merits an increase in the number of "wire channels," as seen in Figure 6. An array of 8 rows has wires up to a nominal length of 16, and one global wire pair (wires of infinite length). An array of 16 rows adds wires with nominal length 32, and one more global wire pair. Each successive doubling adds three new "wire channels," one of which is a global wire pair. An array of 64 rows has wires with nominal lengths up to 128, as well as 4 global wire pairs.

At each logic block, every vertical wire to which the block could connect has assigned to it a unique index that identifies the wire from that logic block. A configuration uses these indices to specify the vertical wires to which a block connects. The assignment of indices to wires is based on a peculiar *twisting* of the wires illustrated in Figure 7. (This twisting gives the vertical wires a recursive structure, a property which can be exploited to improve the efficacy of the *configuration cache* introduced in Section 3.1.4.[1]) Numbers are assigned to wires according to how close the wire is to the logic block in Figure 7. The closest wire is assigned index number 0, the next closest number 1, and so on. Note that, because of the twisting, a wire's number may change from one logic block to another. The assignment of indices is different for each logic block.

There can be at most one driver for each V wire. Configurations are checked when they are first loaded to ensure that this requirement is met. Configurations failing this test cannot be loaded.

2.1.2 Global horizontal wires (G wires)

Unlike the vertical wires, which are always associated with only a single column of array blocks, the G and H wires exist *between* rows and are accessible by logic blocks in the rows above and below the wires (Figure 8). A horizontal wire can be read from both above and below the wire, but it can be driven only by a logic block in the row above the wire. Thus, a horizontal wire can be used to communicate among the columns of a single row, or from a logic block in one row to a different column in the row immediately below. This bias favors computations that proceed downward from one row to the next.

The G wires are the ones in Figure 8 with nominally infinite length. A G wire can be driven by any logic block in the row above the wire. As with the V wires, a configuration that has more than one driver for a G wire cannot be loaded.

Although Figure 8 shows the G wires as spanning the control blocks in the leftmost column of the array, control blocks cannot exam-

[1] How this recursive pattern can be exploited is not covered by this document because such cache implementation details are transparent to a proper processor architecture.

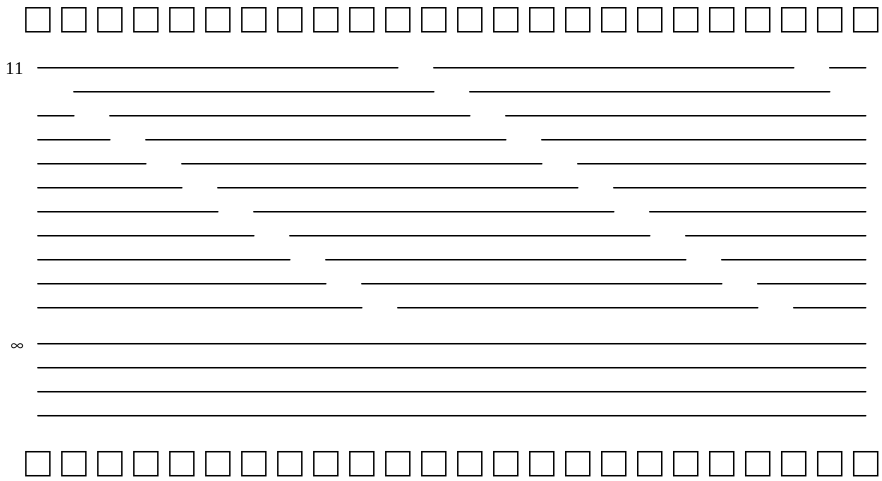


Figure 8: The horizontal wires between two rows. Again, each line actually represents a pair of wires (2 bits). There is a full set of pairs of length 11 (H wires), and four 2-bit buses that span the entire width of the array (G wires). Each wire pair can be read by all of the blocks it spans horizontally, from logic blocks both above and below the wires.

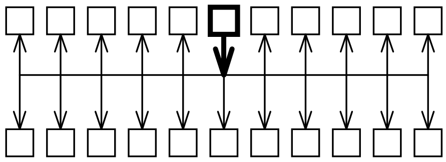
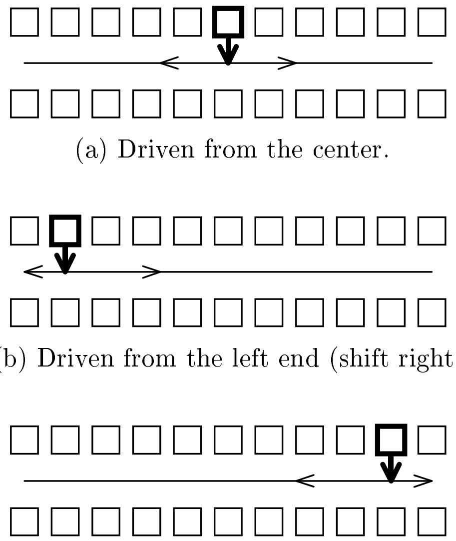


Figure 9: The logic blocks reachable via an H wire driven from the center.



(a) Driven from the center.

(b) Driven from the left end (shift right).

(c) Driven from the right end (shift left).

Figure 10: The three options for driving the H wires below a row.

ine or drive the G wires (Section 3.2).

2.1.3 Local horizontal wires (H wires)

The remaining wires in Figure 8 are H wires, all with nominal length 11. Like the G wires, each H wire can be driven by a logic block from above the wire and can be read by any block above or below the wire. Figure 9 shows the logic blocks reachable via an H wire when the wire is driven by the logic block centered above the wire.

Unlike the other two wire categories (V and G), the H wires are unique in that there are limited options for choosing which logic block drives each H wire. For each row, a single choice is made that determines a unique driver for *all* the H wires immediately below that row. Figure 10 illustrates the three options available to each row. The default is for every H wire below the row to be driven from the center as in Figure 9. Alternatively, every H wire below the row can be driven from near the left end of the wire (Figure 10(b)); or every H wire below the row can be driven from near the right end of the wire (Figure 10(c)).

Which of the three options will be used for driving the H wires across an entire row is determined by the control block at the end of the row. Because the choice of driver is made for all wires along a row in concert, every H wire always has exactly one driver. It is not possible for a configuration to specify more than one driver for any H wire.

2.2 Logic block configurations

The principle data paths within a logic block are depicted in Figure 5. A logic block selects up to four 2-bit inputs, A, B, C, and D, from among the wires at hand, and performs a logical or arithmetic function on these inputs to generate the output value Z. This value is optionally buffered in an internal register and then driven onto as many as three adjacent wire pairs leading to other logic blocks. At the same time, the original D input can also be optionally buffered and driven onto any of the same wire pairs.

A logic block can drive output values simultaneously onto exactly one of the V wire pairs, plus one of the G wire pairs, plus one of the H wire pairs. It is not possible for a single logic block to drive more than one V wire pair, more than one G wire pair, or more than one H wire pair. A logic block can drive any one (or none) of the V wire pairs at hand, and can drive any one (or none) of the G wire pairs below the block (but not above). As stated in the previous section, every logic block drives one H wire pair below, in a pattern across each row that is selected by the control block at the end of the row. In each direction (V, G, and H), the output can be selected from either the Z or the D result.

For each logic block, 64 bits of internal *configuration state* determine the active configuration of that block. The configurable elements of a logic block include the sources of the inputs, the function performed on those inputs, the operation of the registers, and the destinations for the outputs. Figures 11 through 13 detail the encodings of a logic block's configuration state.

Figure 5 shows that a logic block's registers can be read from or written to the memory buses. However, this path is not under the control of the logic block itself and so is not represented in the logic block configuration.

64	58	56	50	48	42	40	34 32
A in	A′	B in	B′	C in	C′	D in	mx

32	16	13	12	11	10	9	8	5 0
lookup table(s)	mode	Z	D	H	G	V	G out	V out

Figure 11: Logic block configuration encoding. 64 bits of configuration state are needed for each active block. The A′, B′, C′, mx, lookup table, and mode fields together determine the logic block function (Section 2.3).

[63..58] A in	
000000	$A = 00$ (binary)
000001	$A = 10$ (binary)
000010	$A =$ internal Z register
000011	$A =$ internal D register
010000	$A =$ V wire pair 15
⋮	⋮
011111	$A =$ V wire pair 0
100000	$A =$ leftmost H wire pair above
⋮	⋮
101010	$A =$ rightmost H wire pair above
101100	$A =$ G wire pair 3 above
⋮	⋮
101111	$A =$ G wire pair 0 above
110000	$A =$ leftmost H wire pair below
⋮	⋮
111010	$A =$ rightmost H wire pair below
111100	$A =$ G wire pair 3 below
⋮	⋮
111111	$A =$ G wire pair 0 below

Figure 12: Configuration of logic block inputs. The four inputs, A, B, C, D, have identical encodings.

[12] Z	
0	suppress latching of Z register; output Z directly
1	latch Z register every cycle; output Z register

[11] D	
0	suppress latching of D register; output D directly
1	latch D register every cycle; output D register

[10] H	
0	$Hout = Z$
1	$Hout = D$

[9] G	
0	$Gout = Z$
1	$Gout = D$

[8] V	
0	$Vout = Z$
1	$Vout = D$

[7..5] G out	
000	no output to G wires below
100	output $Gout$ to G wire pair 3 below
$\vdots$	$\vdots$
111	output $Gout$ to G wire pair 0 below

[4..0] V out	
00000	no output to V wires
10000	output $Vout$ to V wire pair 15
$\vdots$	$\vdots$
11111	output $Vout$ to V wire pair 0

Figure 13: Configuration of logic block registers and outputs.

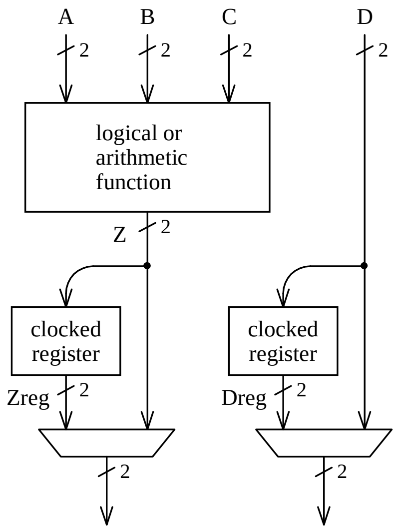


Figure 14: Use of the D input as a completely separate path for routing or copying.

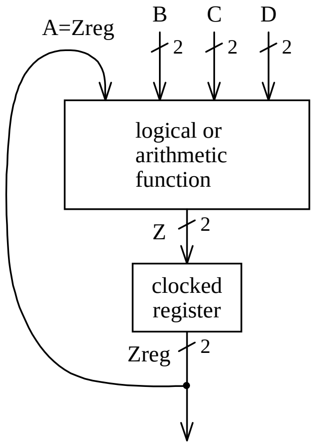


Figure 15: Any of the A, B, C, or D inputs can be taken from the internal registers.

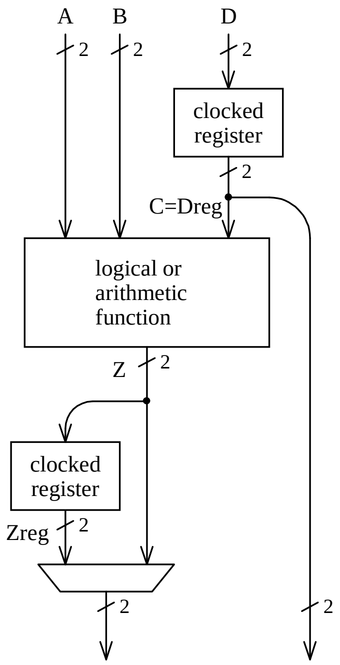


Figure 16: Delaying one logic block input using the D path.

Transfers over the memory bus are instigated by the main processor and/or by the control block at the end of each array row (Section 3).

Note that if the logic block function does not require all four inputs, the D path can be used as a completely independent path—for example to route and/or buffer a value between wires (Figure 14). Many of the available logic block functions ignore the D input, leaving it free for this purpose.

In addition to selecting among the internal wires, any of the A, B, C, or D inputs can be set to a constant. The supported constant values are binary 00 and 10. Binary values 01 and 11 are not provided because they can be synthesized internally by most of the logic block functions. (These functions are covered in the next section.)

The outputs of the internal registers can also be connected back as logic block inputs, as is illustrated for the Z register in Figure 15. A notable application of this feature is to use the D path to buffer an input or the function output for an extra cycle. Figure 16 shows how an input can be delayed by connecting the D register output to one of the function inputs. Conversely, in Figure 17 the D path is tied to the Z register output to delay the Z result one cycle.

Each register can operate in either of two modes. If a register is used as a buffer, it automatically latches a new value every clock cycle. Alternatively, a register can be bypassed on output, in which case it never latches except when it is written to via a memory bus. The input of a bypassed register is thus effectively decoupled from the logic block. Note that this implies, for instance, that the registers in the examples of Figures 15 and 17 could not be bypassed on output, or else they would cease to act as buffers.

From the perspective of the memory buses, a bypassed register will hold a value written to it until it is updated again over a memory bus. By connecting the output of such a register to a logic block input, a value latched from a memory bus can be used immediately in the logic block function. Figure 18 demonstrates the use of both internal registers for holding memory bus inputs in this way. Either register can also be directly output via the D path, if desired.

A register selected as a buffer can also be written to over a memory bus, in which case the memory bus value supercedes any internal value for that clock cycle. The value written from the memory bus will be subsequently overwritten in the next clock cycle.

Figure 19 gives a more detailed view of the logic block internal paths.

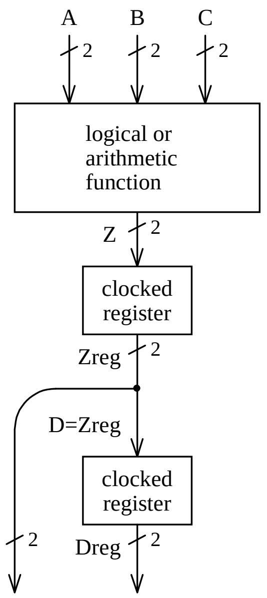


Figure 17: Delaying the Z output using the D path.

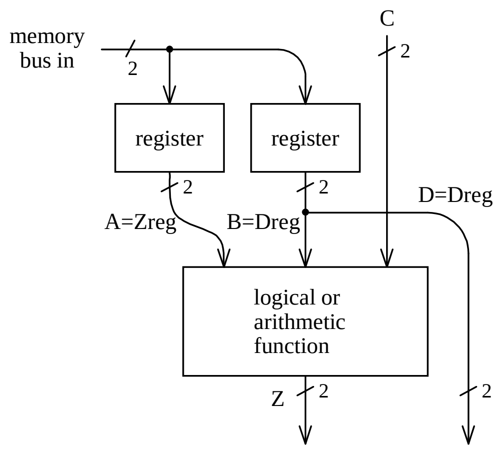


Figure 18: Values read over the memory buses can be latched into either internal register and used immediately as function inputs within the logic block.

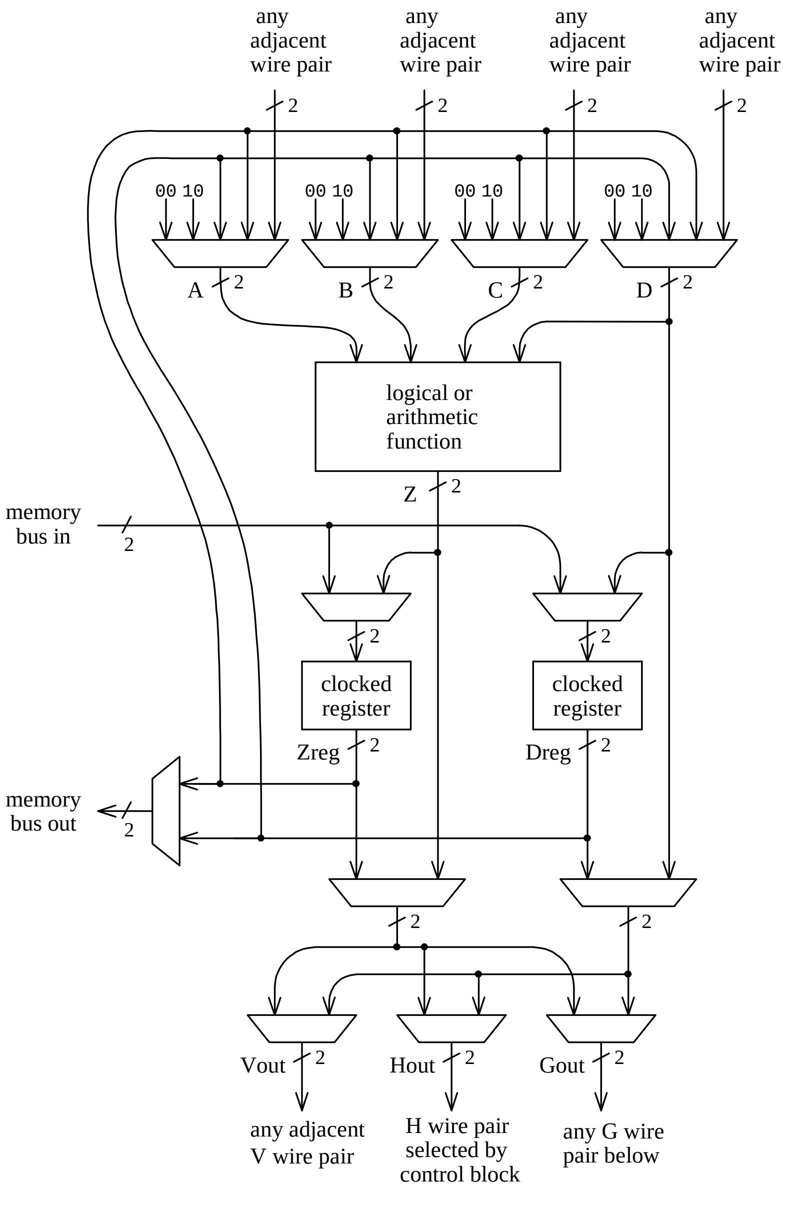


Figure 19: Logic block internals.

mode [15..13]	mx [33..32]		
000	D′	table mode	
001	00	fifth input table mode	
001	01	split table mode	
01k	00	select mode	(k = 0 suppresses shifts in)
01k	01	partial select mode	”
01k	10	variable shift mode	”
10k	result	carry chain mode	(k = 0 suppresses carry in)
11k	result	triple add mode	(k = 0 suppresses shifts, carries in)

Figure 20: The function mode encodings. For many modes, bit k (bit 13) determines whether shifts and carries in are to be suppressed.

2.3 Logic block functions

This section details the computational functions that a logic block can perform. The main inputs to this function are the four values A, B, C, and D, each of which is 2 bits in size. The primary output is Z, also 2 bits.

In addition to the primary ones, several miscellaneous inputs and outputs are associated with specific logic block functions. Most of these extra connections are to a block's nearest leftmost and rightmost neighbors to support multi-bit functions built out of multiple logic blocks along a row. The H wires above a logic block are also taken as extra inputs for variable shifts. Details about these extra inputs and outputs are given below as each becomes relevant.

A logic block's function is determined by several fields in the active configuration—the A′, B′, C′, mx, lookup table, and mode fields (Figure 11). The mode and mx fields together select among the eight possible function modes (Figure 20). Each mode is defined below. Most modes make some use of the lookup table, although some do not. In all modes, the A′, B′, and C′ fields choose some form of initial perturbation of the corresponding inputs.

2.3.1 Table mode

Table mode is the basic mode for performing simple logical functions, as shown in Figure 21. After each input passes through a *crossbar* function, a table lookup implements an arbitrary bitwise logical operation on the four inputs to give the result. Table mode is selected when the configuration's mode field is 000 (binary).

The function of the crossbars is illustrated in Figure 22. As its name implies, each crossbar allows each of its 2 output bits to be selected independently from either of its input bits. There are four possibilities: pass the incoming bits through unperturbed, duplicate incoming bit A_0, duplicate incoming bit A_1, or swap the two bits. As there is no D′ configuration field, the mx field selects the D crossbar function in this mode.

The 16-bit lookup table specifies an arbitrary 4-input logical function f, as shown in Figure 23. This function is independently applied to the high and low bits of the four inputs to generate the high and low bits of the result; i.e., $Z_1 = f(A'_1, B'_1, C'_1, D'_1)$ and $Z_0 = f(A'_0, B'_0, C'_0, D'_0)$.

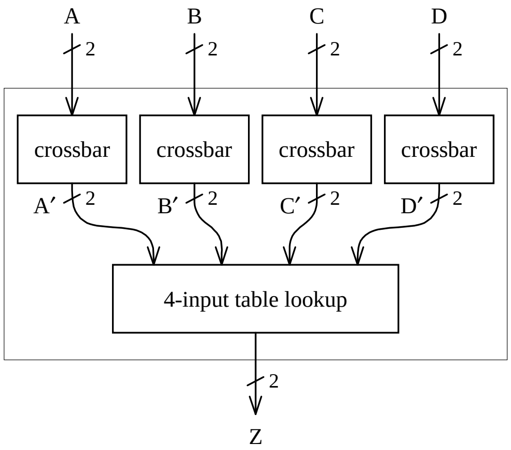
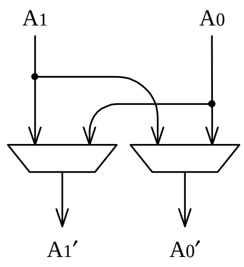


Figure 21: Table mode (mode = 000). The mx field selects the crossbar function for D'.



[57..56] A′	
00	$A' = A_0 A_0$
01	$A' = A_0 A_1$
10	$A' = A_1 A_0$
11	$A' = A_1 A_1$

Figure 22: The crossbar functions.

2.3.2 Fifth input table mode

Fifth input table mode (Figure 24) is identical to table mode, except that the D' input is set to the value of *Hout* from the logic block in the same column in the row above. This is the value being driven onto the H wires by the logic block immediately above. (There is always some such value, with the exception of the first row of a configuration.) The usual D path through the logic block is not affected (Section 2.2); fifth input table mode simply ignores the D input to the logic block function. The mode provides a fourth input for calculating Z independent of D.

Fifth input table mode is selected when mode = 001 and mx = 00. The mode is illegal on the topmost row of a configuration.

2.3.3 Split table mode

Split table mode (Figure 25) is again just like table mode, except that for this mode D' is fixed at 10 (binary). This has the simple effect of allowing the two bits of Z to be calculated using separate 3-input functions, as shown in Figure 26. Split table mode is chosen when mode = 001 and mx = 01.

32 16

Z bit

1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 $\longleftarrow A'$ bit
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 $\longleftarrow B'$ bit
1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 $\longleftarrow C'$ bit
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 $\longleftarrow D'$ bit

Figure 23: Interpretation of the lookup table in table mode and fifth input table mode. The lookup table function f takes 4 input bits and returns a single output bit. Listed underneath each table entry is the pattern of input bits corresponding to that table output. The 2 bits of Z are calculated independently using the same function: $Z_1 = f(A'_1, B'_1, C'_1, D'_1)$ and $Z_0 = f(A'_0, B'_0, C'_0, D'_0)$.

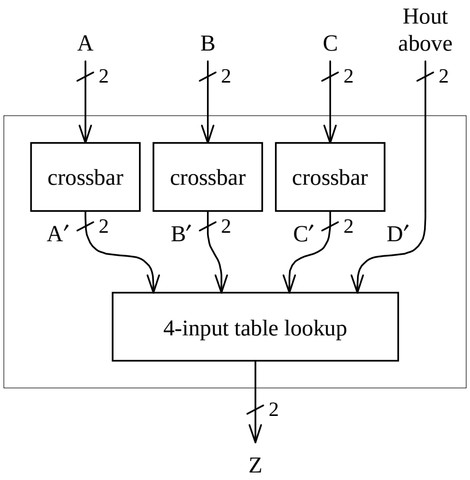


Figure 24: Fifth input table mode (mode = 001, mx = 00). "Hout above" is the value driven onto the H wires by the logic block immediately above in the same column.

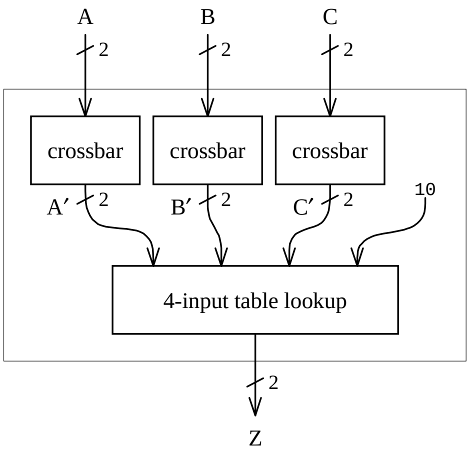


Figure 25: Split table mode (mode = 001, mx = 01).

32 24

Z_1

1 0 1 0 1 0 1 0 $\longleftarrow A'_1$
1 1 0 0 1 1 0 0 $\longleftarrow B'_1$
1 1 1 1 0 0 0 0 $\longleftarrow C'_1$

24 16

Z_0

1 0 1 0 1 0 1 0 $\longleftarrow A'_0$
1 1 0 0 1 1 0 0 $\longleftarrow B'_0$
1 1 1 1 0 0 0 0 $\longleftarrow C'_0$

Figure 26: Interpretation of the lookup table in split table mode. Forcing $D'_1 = 1$ and $D'_0 = 0$ causes the 2 bits of Z to be calculated based on separate 3-input functions. Compare with Figure 23.

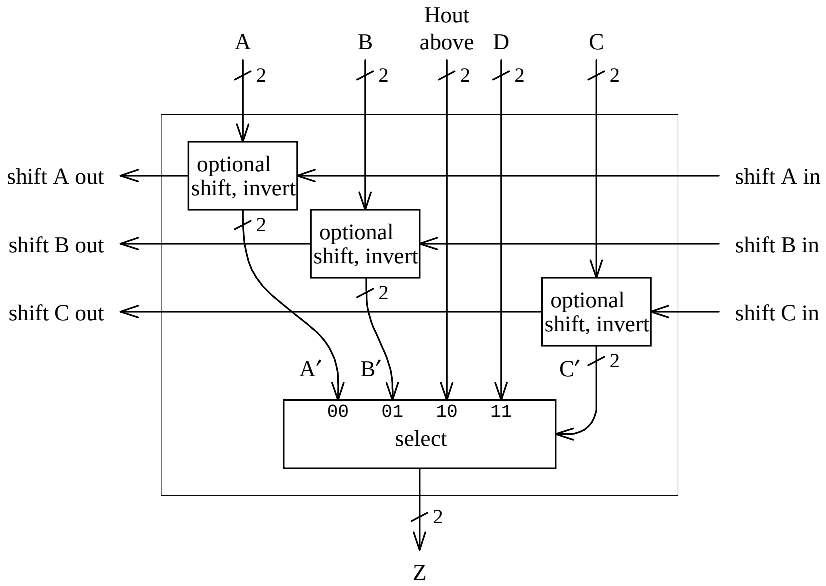


Figure 27: Select mode (mode = 011, mx = 00). If mode bit 0 is set to 0 (i.e., mode = 010, mx = 00), the function is the same except that all shifts in are assumed to be 0.

2.3.4 Select mode

Select mode implements a multiplexor of four inputs, as illustrated in Figure 27. In place of crossbar functions, two of the multiplexor inputs, A and B, and also the controlling input C, are first optionally shifted and/or complemented (inverted) to form the perturbed values A', B', and C'. The resulting C' is then used to select one of the other four inputs as follows:

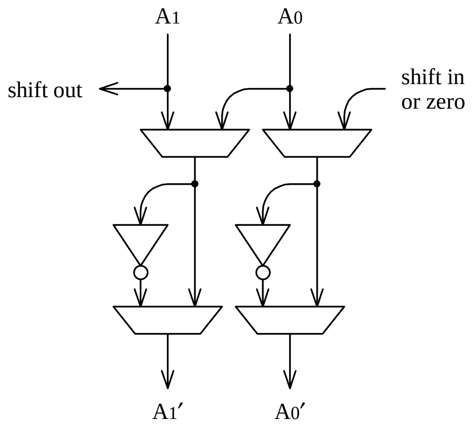
C'	Z
00	A'
01	B'
10	*Hout* from row above
11	D

As in fifth input table mode, one of the inputs is the value of *Hout* from the logic block in the same column in the row immediately above. It is illegal for C' to be 10 (binary) if select mode is used on the topmost row of a configuration.

The functions of the shift-invert blocks are shown in Figure 28. A 2-bit input is first optionally shifted left one bit, and then the resulting 2-bit value (shifted or not) is optionally complemented. When a shift is performed, a bit to shift in is taken from the high bit of the same input from the logic block to the immediate right (regardless of what mode the logic block on the right is in). It is illegal to depend on the bit shifted into the rightmost logic block on a row.

The shifts into all three shift-invert blocks can together be forced to 0 by the configuration. This option is useful for the rightmost logic block of multi-block functions and also for the rightmost logic block on a row. Shifts into the individual shift-invert blocks cannot be independently suppressed.

Select mode is chosen with mode = 010 or



[57..56] A′	
00	$A' = A_1 A_0$
01	$A' = \neg(A_1 A_0)$
10	$A' = A_0 A_{-1}$
11	$A' = \neg(A_0 A_{-1})$

Figure 28: The optional shift-invert functions. The "¬" symbol represents logical negation (inversion). If the function mode suppresses shifts in, bit A_{-1} is 0. Otherwise, bit A_{-1} is taken from bit A_1 from the logic block on the right (regardless of the mode the logic block on the right is in).

011, and mx = 00. The first case (mode = 010) suppresses shifts in, while the second (mode = 011) does not.

Select mode performs no table lookups. The configuration's lookup table field must be set to the constant

32															16
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

2.3.5 Partial select mode

Partial select mode (Figure 29) is identical to ordinary select mode except that the set of inputs is different:

C'	Z
00	A'
01	B'
10	B (not shifted or inverted)
11	00

This mode is enabled when mode = 010 or 011, and mx = 01. Setting mode to 010 suppresses perturbation shifts in, while mode = 011 does not.

In partial select mode there is no restriction on the value of C' on the topmost row of a configuration (in contrast to ordinary select mode).

2.3.6 Variable shift mode

Variable shift mode supports the implementation of variable shifts across a row (Figure 30). As with the select modes, the B and C inputs are first passed through shift-invert perturbation functions. The resulting B' and C' values are then used to select two bits from among the H wires above. The A and D inputs are ignored.

The 11 pairs of H wires are treated as a single 22-bit value v, with the physically leftmost wire pair contributing the most significant two bits v_{21} and v_{20}, and the physically rightmost

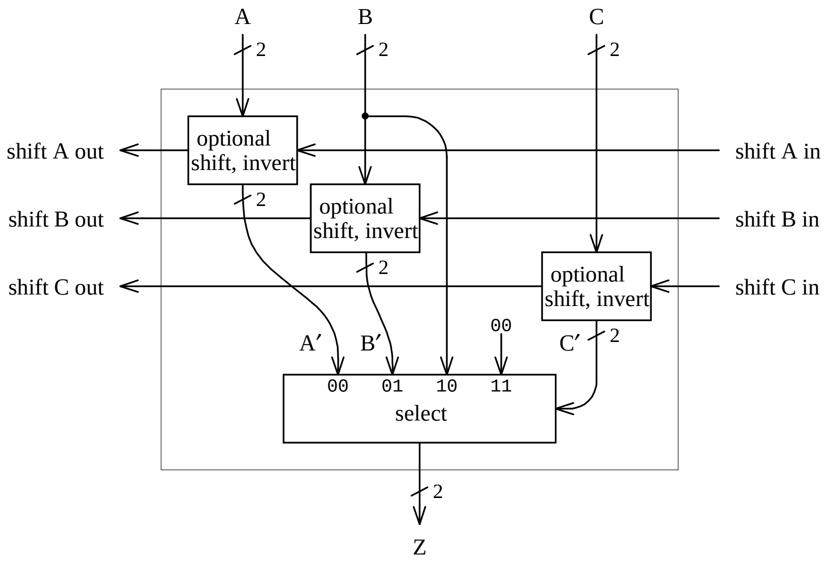


Figure 29: Partial select mode (mode = 011, mx = 01). Again, if mode bit 0 is set to 0 (mode = 010, mx = 01), the function is the same except that all shifts in are assumed to be 0.

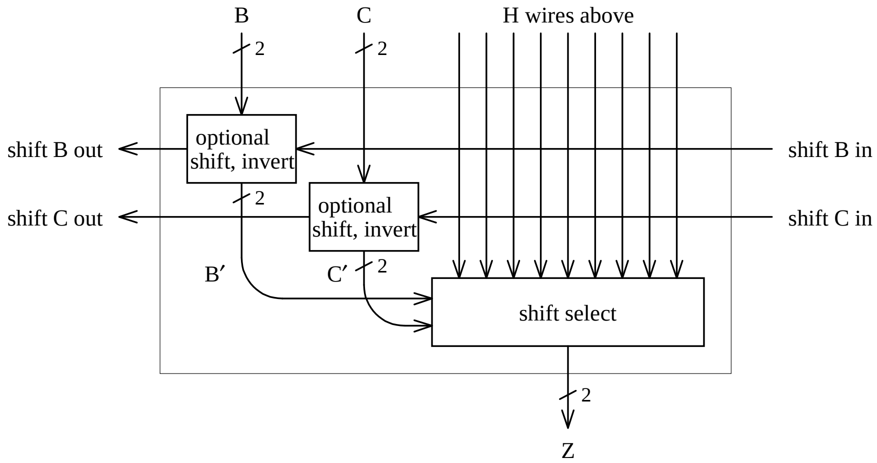


Figure 30: Variable shift mode (mode = 011, mx = 10). If mode bit 0 is set to 0 (mode = 010, mx = 10), the function is the same except that shift B in and shift C in are assumed to be 0.

wire pair contributing the least significant two bits v_1 and v_0. *If nonzero*, the 4-bit concatenation $B'_1B'_0C'_1C'_0$ acts as an index into this 22-bit integer. Given $B'_1B'_0C'_1C'_0 = i \neq 0$, bits v_{19-i} and v_{18-i} are selected to give Z.

In the special case that the index $B'_1B'_0C'_1C'_0$ is zero, Z is set to *Hout* from the logic block in the same column in the row immediately above (regardless of *which* pair of wires the logic block above is actually driving).

Variable shift mode is selected when mode = 010 or 011, and mx = 10. In the case of mode = 010, shift B in and shift C in are forced to 0.

Variable shift mode is illegal on the topmost row of a configuration. Also, as for the select modes, this mode performs no table lookups, and the lookup table field must be set to the constant

32															16
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

2.3.7 Carry chain mode

Carry chain mode performs a logical or arithmetic function involving the fast carry chain across a row. The mode is diagramed in Figure 31. Only three inputs, A, B, and C, are used; the D input is ignored. (The D path still exists and can be employed separately; see Section 2.2.) Like table mode, the three inputs are passed through crossbar functions before being applied to table lookups. The table lookup results are used to control the carry chain, and then these same values are logically combined with the carry chain output to obtain the final result Z.

The table lookups deliver a total of four bits that control the carry chain: a *propagate* and a *generate* signal are associated with the low bit of the result, and another pair of such signals are associated with the high bit of the result. Figure 33 shows how these control bits affect the carry chain. If a *propagate* bit is 1, the carry into that position is propagated to the next higher bit position; otherwise, the corresponding *generate* value is used as the carry out to the next bit position. When *propagate* is 1, the *generate* value is ignored by the carry function (although it may still be used in the result function; recall Figure 31). As Figure 33 shows, the carry chain repeats the same operation at each bit position.

The operation of the lookup tables is documented in Figure 32. There are two 3-input tables: one is the *propagate* table, and the other the *generate* table. Each table is looked up twice, once for the low bit position and once for the high bit position.

The carry chain outputs a 2-bit value K, comprising the carry into each bit position (Figure 33). This is fed into the result function, along with the original *propagate* and *generate* signals, which are renamed to U and V, respectively. The result function implements one of four bitwise logical functions given in Figure 34, chosen by the mx field of the configuration.

A logic block is in carry chain mode when mode = 100 or mode = 101. The first case forces the carry into the low bit (K_0) to be 0. The second case accepts the carry out from the logic block on the right.

It is illegal to depend on the carry in when the logic block to the immediate right is not in carry chain mode. Likewise, it is illegal to depend on the carry into the rightmost logic block on a row.

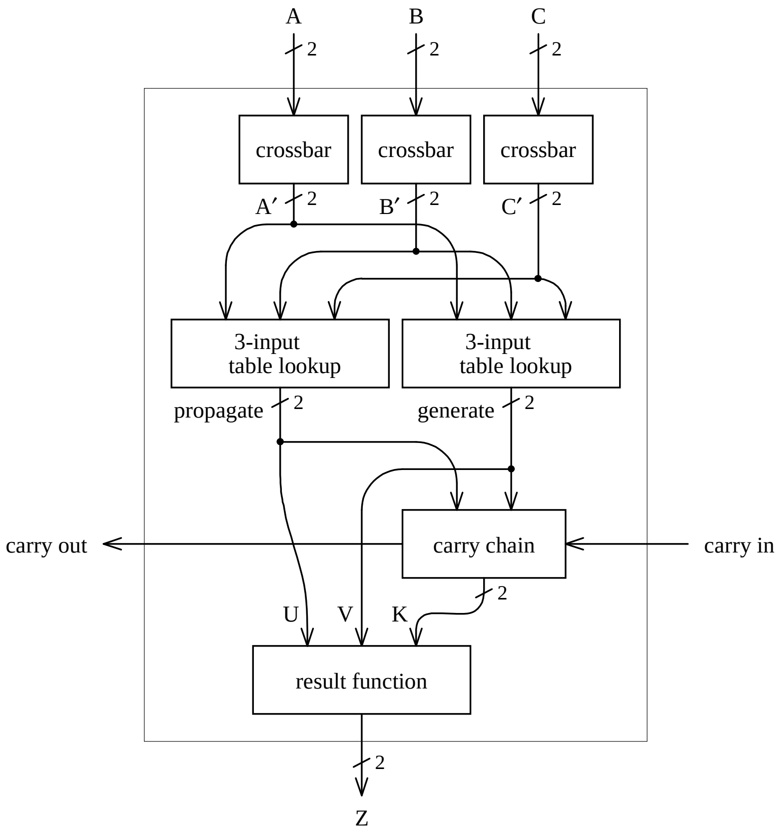


Figure 31: Carry chain mode (mode = 101). The mx field selects the result function. If mode bit 0 is set to 0 (i.e., mode = 100), the function is the same except that the carry in is assumed to be 0.

32 | U bit (propagate) | 24

1 0 1 0 1 0 1 0 ⟵ A' bit
1 1 0 0 1 1 0 0 ⟵ B' bit
1 1 1 1 0 0 0 0 ⟵ C' bit

24 | V bit (generate) | 16

1 0 1 0 1 0 1 0 ⟵ A' bit
1 1 0 0 1 1 0 0 ⟵ B' bit
1 1 1 1 0 0 0 0 ⟵ C' bit

Figure 32: Interpretation of the lookup table in carry chain mode.

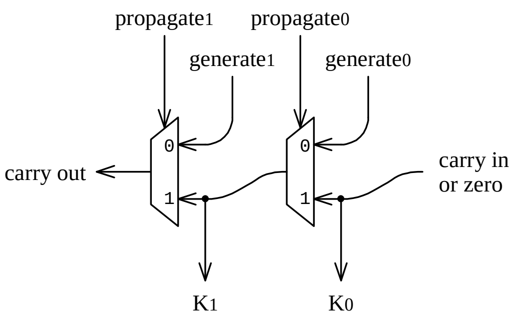


Figure 33: Operation of the carry chain. The low-order carry in can be forced to 0 by the logic block mode.

[33..32] mx	
00	$Z = V$
01	$Z = V \vee K$
10	$Z = U \oplus K$
11	$Z = \neg(U \oplus K)$

Figure 34: The result functions for modes using the carry chain.

2.3.8 Triple add mode

The most complex mode is triple add mode, which can perform a sum or difference of three inputs (Figure 35). Each of the three inputs is first passed through a shift-invert function (Figure 28), and then a carry-save addition is performed on the perturbed inputs. The two outputs of the carry-save addition are used to index two lookup tables to obtain the carry chain *propagate* and *generate* signals (Figure 36). From this point forward, triple add mode is identical to the simpler carry chain mode.

The carry-save addition performs the usual function, with a so-called "*sum*" output calculated bitwise as $A' \oplus B' \oplus C'$, and a *carry* output calculated as $(A' \wedge B') \vee (A' \wedge C') \vee (B' \wedge C')$ shifted left by one bit position in the same manner as the shift-invert functions. As with the shift-invert functions, the shift carry in (ultimately $carry_0$) can be forced to 0 by the configuration mode field.

Triple add mode is selected when mode = 110 or 111. The first case forces the shifts in and the carry in to be 0, whereas the second case accepts these from the logic block on the right. The mx field specifies the result function.

As for carry chain mode, it is illegal to depend on the carry in or the shift carry in when the logic block to the immediate right is not in triple add mode. Likewise, it is illegal to depend on the shifts or carry into the rightmost logic block on a row.

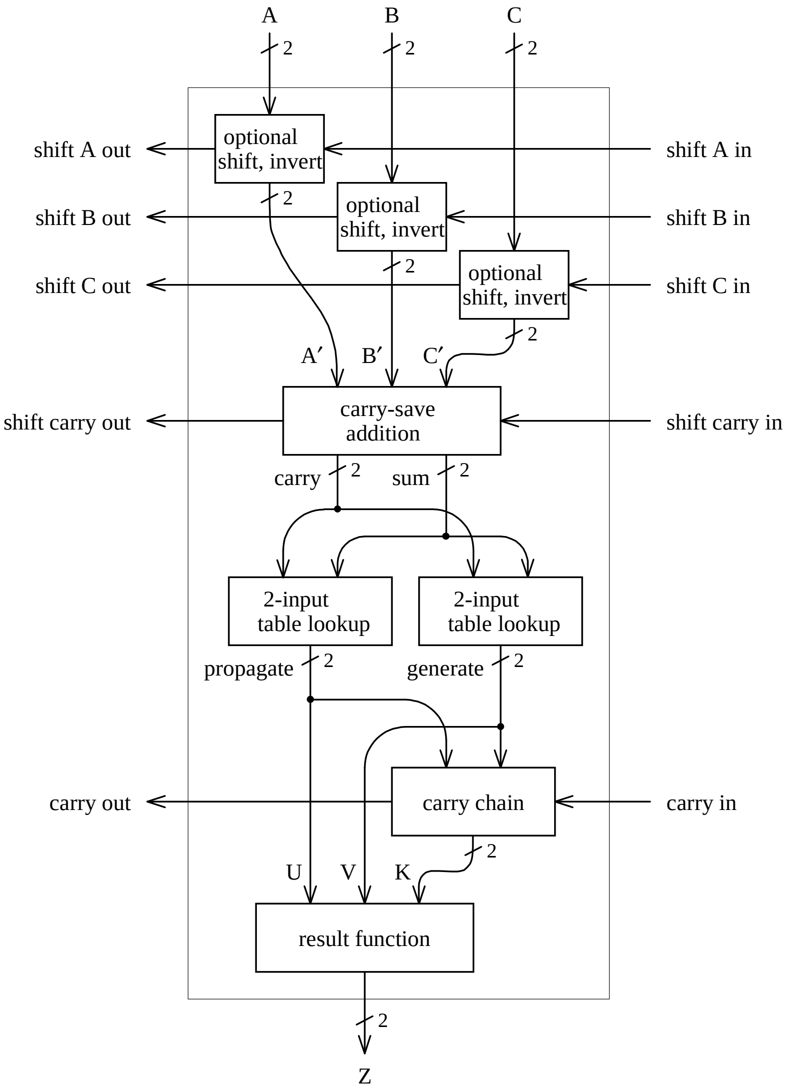


Figure 35: Triple add mode (mode = 111). The mx field selects the result function. If mode bit 0 is set to 0 (i.e., mode = 110), the function is the same except that all shifts and carries in are assumed to be 0.

32 24		24 16	
$\boxed{U \text{ bit (propagate)}}$		$\boxed{V \text{ bit (generate)}}$	
1 0 1 0 1 0 1 0	$\longleftarrow$ carry bit	1 0 1 0 1 0 1 0	$\longleftarrow$ carry bit
1 1 0 0 1 1 0 0	$\longleftarrow$ sum bit	1 1 0 0 1 1 0 0	$\longleftarrow$ sum bit

Figure 36: Interpretation of the lookup table in triple add mode. Each table is represented by eight bits, even though four bits would be sufficient. The redundancy in the tables is required and must be consistent.

2.4 Internal timing

Delays within the array are defined in terms of the sequences that can 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. A *short wire* is a local horizontal wire (H wire) or a vertical wire of length 8 or less. A *simple function* is either a table mode function or a traversal of the independent "D path" in a logic block. At the end of a cycle, values can be latched in logic block registers without affecting these rules.

Within combinatoric circuits, it is not necessary to latch intermediate results in registers at the end of every clock cycle unless the latches are desired to achieve pipelining. However, there is a maximum allowed path delay between registers of 8 array clock cycles.

3 Integration of array with main processor

The loading and execution of array 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. Configurations and data are transferred to/from the array over the memory buses that run through the entire array (Figure 2).

During array execution, the array itself can initiate reads or writes to memory (via the memory buses) without intervention by the main processor. Such memory accesses are coordinated by the control blocks at the end of each array row. Array memory accesses go through the same memory hierarchy as the main processor, including the first-level data cache. The array thus has available to it a relatively large, fast memory store which is automatically kept consistent with memory accesses made by the processor.

In addition to on-demand "random" accesses to memory, three *array memory queues* provide enhanced support for sequential memory accesses.

3.1 Processor control of array

3.1.1 Array clock counter

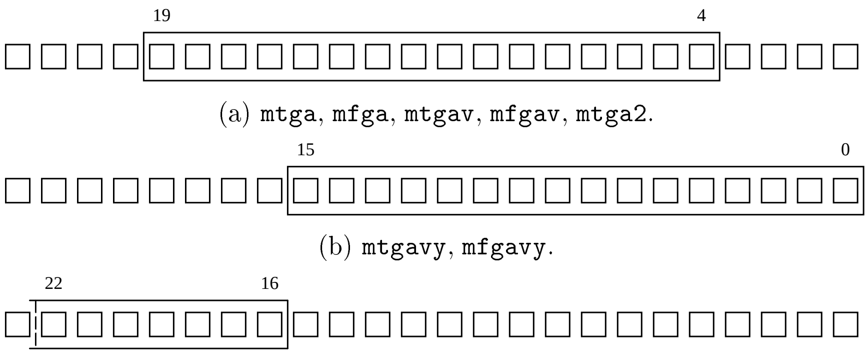
Array execution is governed by a countdown counter called the array clock counter. While the clock counter is nonzero, it is decremented by 1 with each array clock cycle. When the array clock counter is zero, the latching of array registers is disabled, effectively stopping the array.

A configuration can be loaded into the array only when the clock counter is zero. After loading a configuration, the main processor can set the array clock counter to nonzero to start the array executing for a given number of clock cycles. The counter can be set using the `gabump` instruction detailed in Figure 38(a). Various other processor instructions are also able to set the counter in addition to their other functions.

There is no defined relationship between the array clock and the rate at which the main processor executes instruction. To ensure proper synchronization, most processor instructions that interact with the array first stall until the clock counter reaches zero before performing their function. The clock counter thus provides the mechanism by which array calculation delays are interlocked with subsequent dependent processor instructions.

Since the number of clock cycles needed for a calculation may not be known in advance, the array has the ability to halt itself whenever its function is complete, by forcibly zeroing the clock counter. How the array can be configured to do this is discussed along with the other functions of the control blocks in Section 3.2. It is also possible for the processor to halt the array at any time by zeroing the clock counter. The `gastop` instruction which performs this function is covered in connection with context switches in Section 3.1.6.

The array clock counter is a 32-bit register, of which only the least significant 31 bits actually count down. The most significant bit is a "sticky" bit: once set, it remains set until the entire counter is forcibly zeroed either by the array or by the processor (`gastop`). Since the array is halted only when the entire 32-bit counter is zero, the most significant bit acts as an "infinity" bit. If the latency of an array calculation is entirely data-dependent, the processor can set the most significant bit of the clock counter to start the array operating indefinitely. The array can then zero the clock when its computation completes. If the processor is ready to receive array results be-



(a) `mtga`, `mfga`, `mtgav`, `mfgav`, `mtga2`.

(b) `mtgavy`, `mfgavy`.

(c) `mtgavz`, `mfgavz`.

Figure 37: The set of logic blocks read or written by various processor instructions. The `mtga` (*move to Garp array*) instructions copy a 32-bit word from a processor register to a contiguous set of logic block registers along an array row. The `mfga` (*move from Garp array*) instructions transfer a word in the opposite direction. Since logic block registers are 2 bits each, 16 logic blocks correspond to a 32-bit data word. The leftmost block on each row is a control block which contains no visible data registers.

fore the array is done, the first instruction attempting to retrieve data from the array will interlock as usual until the counter is zeroed.

3.1.2 Transfering data to/from array

The processor has a collection of instructions for copying data between the processor register file and the registers in array logic blocks. Since a single logic block's Z or D register is only 2 bits, most data transfers gang together 16 contiguous logic blocks on a row so that 32 bits of data are copied at a time. An individual transfer copies to or from the 16 combined Z registers or the 16 combined D registers of the 16 logic blocks. For each transfer operation, an array row number must be specified, along with whether the array source/target is to be the Z or D registers. These two parameters (row number and Z/D selection) are encoded as constants within some instructions, while other instructions obtain them from an additional register operand.

Each row has 23 logic blocks, but an individual transfer operation only touches at most 16 of them corresponding to a full 32-bit word. The set of logic blocks read or written is fixed for each particular instruction (Figure 37). Most instructions copy to/from only the middle 16 logic blocks found in columns 4 through 19 inclusive. A few variant instructions allow access to the logic blocks at the extreme left or right ends of a row. The rightmost logic block is always associated with the least significant 2 bits transferred, and the leftmost logic block is associated with the most significant 2 bits.

The instructions for copying data to/from the array are detailed in Figures 38(b)–(e). The `mtga` (*move to Garp array*) instruction transfers a 32-bit word from a processor register to the Z or D registers of the middle 16 logic blocks of a fixed row. The row number and the choice of Z or D registers are encoded as constants in the `mtga` instruction. The `mfga` (*move from Garp array*) instruction is the same except it transfers in the opposite direction, from the array to a processor register. Instructions `mtgav` and `mfgav` are similar, but instead of hardcoding the array row number and Z/D register choice in the instruction, a second register operand supplies these parameters.

Access to the logic blocks in the leftmost and rightmost columns of a row is provided by variants of `mtgav` and `mfgav`. The `mtgavy` and `mfgavy` instructions access columns 0 through 15 but are otherwise identical to `mtgav` and `mfgav`. At the other end of a row, variants `mtgavz` and `mfgavz` read or write the 14 bits of columns 16 through 22. (Column 23 is left out because it contains the control blocks, which have no visible data registers.) These last two instructions are unusual in that they only transfer 14 bits. For `mtgavz`, the most significant 18 bits of the source processor register are ignored; while in the other direction, `mfgavz` zeros the most significant 18 bits of the destination processor register.

Unlike the other transfer instructions, `mtga2` can transfer more than one word at a time. In one operation, `mtga2` copies two 32-bit processor registers to the middle logic blocks of two or more independent array rows. Both values can be copied to multiple destination rows at once. The destination rows are not encoded in the instruction but are instead selected by the array itself based on a *match code* encoded in the `mtga2` instruction. Additional explanation can be found in Figure 38(e) and Section 3.2.1.

The processor can copy to or from the array only when the array clock counter is zero. If the clock counter is nonzero, a data transfer instruction will stall until the clock counter becomes zero. The instructions `mtga`, `mfga`, and `mtga2` can also set the clock counter to a small constant after performing their transfer.

3.1.3 Array condition flag

The array has a 1-bit *condition flag* that can be examined by the main processor. Ordinarily the array sets this flag to indicate some condition to the main processor. Two processor branch instructions (Figure 38(f)) are based on the array condition flag: `bgat` and `bgaf`. The `bgat` instruction branches if the condition flag is set (*true*), whereas `bgaf` branches if the flag is clear (*false*).

3.1.4 Loading configurations

The loading of array configurations is under the control of instructions executed by the main processor. Loading a configuration makes the configuration *active*, so that the configuration controls the behavior of the array. Only one configuration can be active in the array at a time. Loading a new configuration replaces the previous one.

Although logically only one configuration can be loaded at a time, in practice one can expect an actual implementation to incorporate within the array a *configuration cache* of recently loaded configurations, so that the process of "loading" a configuration does not necessarily involve transferring it from external memory every time. Only a few processor clock cycles should be needed to load a configuration from the configuration cache. If a configuration is not in the cache, it can be expected that close to the full aggregate bandwidth of the memory buses will be used to load it from external memory.

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. The first, topmost row of a configuration is row number 0 by default, and subsequent rows are labelled with increasing integers.

The active configuration can be changed only when the array clock counter is zero (the array is halted). The instructions that load configurations will stall waiting for the clock counter to become zero before performing their function.

The simplest instruction for loading configurations is `gaconf`, which takes a single register operand giving the address of the configuration stored in memory. The first 4 bytes (32 bits) at this address are interpreted as a count of the number of rows of the configuration. Following this row count is 8 bytes for each block (control blocks and logic blocks) of the configuration, starting with $24 \times 8 = 192$ bytes for row 0, and so on for each row. The configuration for a row contains first the 8 bytes for the control block, followed by the logic block in the leftmost column 22, on down to the rightmost logic block in column 0. In addition to loading a configuration into the array and making it active, `gaconf` initializes the Z and D registers of all logic blocks to zero.

During the time a configuration is active, its copy in memory must not be changed because it may need to be reloaded at any time. (Reloads can be caused by context switches in a multitasking system, for example.) Furthermore, if an inactive configuration is modified in memory and explicitly reloaded, the changes may not take effect if an earlier unmodified version of the configuration is still in the cache. Before an attempt is made to load a modified configuration, the previous version must be cleared for certain from the cache. The `gacinv` instruction performs this function; and it can be executed even while another configuration is running.

The `gaconf` instruction does not allow state to remain in the logic block registers from one configuration to another. A more complex pair of instructions supports configuration *overlays* for this purpose. The `gaalloc` instruction reserves a group of rows into which subsequent configurations will be overlayed. Like `gaconf`, `gaalloc` displaces any currently active configuration and zeros all of the Z and D registers in the array; but no configuration is yet loaded by the instruction. The `gaconfo` instruction loads a configuration into a previously allocated group of rows. An overlayed configuration may not extend beyond the rows allocated by `gaalloc`, but it need not fill the allocation, and it may be loaded starting at a row other than the first allocated row. All of the register state within the allocated space is preserved from one overlay to another. Nevertheless, if an overlay is smaller than the allocated space, only the rows of the overlaying configuration are made active. Inactive rows hold their values until subsequently made active.

The `gaalloc` instruction takes as an operand a pointer to a 32-bit word in memory, the value of which is the number of contiguous rows to allocate. Although logically this indirection through a pointer is unnecessary (the register operand could just as easily have specified the number of rows directly), the pointer is intended to be used as an identifier internally by the cache. If a later execution of `gaalloc` precedes a repeat sequence of `gaconfo` overlays, the same pointer should be used as the operand to both `gaalloc`'s in order to maximize cache utilitization.

For completeness, `gareset` "unloads" any active configuration. Array activity is disabled until such time as another configuration is loaded.

3.1.5 Memory queue control

Two processor instructions (`galqc` and `gasqc`) are used to load and store the state of the array *memory queues*. The details of these instructions are deferred until Section 3.3 when array memory queues are covered.

3.1.6 Saving and restoring array state

Information about the active configuration is stored in three read-only registers:

`$gacr3` – The pointer that was the argument to `gaalloc` when the current array allocation was made. The 32-bit word at this address gives the number of rows allocated. If the array allocation was made by `gaconf` (without a separate `gaalloc`), this is the pointer that was the argument to `gaconf`.

`$gacr4` – The pointer to the configuration in memory that was the argument to `gaconf` or `gaconfo`.

`$gacr5` – The row offset that was the argument to `gaconfo`. If the active configuration was loaded by `gaconf`, this value is zero.

The `cfga` instruction can be used to retrieve any one of these values into a processor register.

When a context switch occurs while the array is active, it must be possible to suspend the array and save its state so that the computation can be resumed at a later time. The first step toward suspending the array is to execute the `gastop` instruction, which in one step copies the clock counter to a processor register and zeros the counter. The current allocation and configuration can be obtained from the array control registers above, and the logic block registers can be read out using the `mfgav` instructions already described. The state of the array memory queues is saved using the `gasqc` instruction. The remaining internal state of the array, including the status of pending memory reads, can be written to memory using the special `gasave` instruction.

Resuming an array computation requires first that the array allocation be restored by executing `gaalloc` with the previously saved value from `$gacr3`. The active configuration is reloaded by executing `gaconfo` with the values that were saved from `$gacr4` and `$gacr5`. The logic block registers can be restored using simple `mtgav` instructions, while `galqc` reloads the state of the array memory queues. Once the logic block registers are restored, `garestore` can be used to read back the internal state that had been saved by `gasave`. The final step for resuming the array is to use `gabump` to restore the array clock counter to the value originally returned by `gastop`.

Besides reading back the array's internal state, `garestore` also ensures that combinatorial propagations in the array are given time to complete, following the recent restoration of the logic block register values.

Increase array clock counter

32 25 21 16 11 5 0

0 1 0 0 1 1 1	0 0 0 0	0 0 0 0 0	rd	0 0 0 0 1 0	0 0 0 0 0

`gabump` *rd*

Adds the value in register *rd* to the array clock counter. The addition is performed modulo 2^{32}. If a carry out of the most significant bit occurs (unsigned overflow), the most significant bit of the clock counter is set.

Figure 38(a): Instruction for setting or incrementing the array clock counter.

Copy word to array

32 25 21 16 6 5 0

0 1 0 0 1 1 1	1 0 0 1	rt	row number	R	clock count

`mtga` *rt,reg,count*
`mtga` *rt,reg*

Copies the value in register *rt* to the middle 16 logic blocks of a fixed array row. The array row number is encoded as an unsigned integer constant in the instruction. If instruction bit R is 0, the concatenation of the sixteen 2-bit *Z* registers in columns 4 through 19 is the destination of the copy. If R is 1, the concatenation of the sixteen D registers in columns 4 through 19 is the destination. Column 4 receives the least significant 2 bits of the value copied and column 19 receives the most significant 2 bits.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. The copy is then performed, after which the clock counter is set to the 5-bit unsigned integer constant encoded in the instruction.

For the *reg* argument to `mtga`, the assembler accepts the syntax `$z`*n* or `$d`*n*, where *n* is the array row number expressed as a decimal integer numeral. (For example, `$z19` denotes the *Z* registers of array row 19.) The *count* argument must be an integer constant. If *count* is not given it defaults to zero.

Copy word from array

32 25 21 16 6 5 0

0 1 0 0 1 1 1	1 0 0 0	rt	row number	R	clock count

`mfga` *rt,reg,count*
`mfga` *rt,reg*

This instruction is identical to `mtga` except that the direction of the copy is reversed.

Figure 38(b): Instructions for copying a word to/from a specific array row.

Copy word to array, variable row

32 … 25	21	16	11	5	0
0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 0 1 1	0 0 0 0 0

`mtgav` *rt*,*rd*

Copies the value in register *rt* to the middle 16 logic blocks of the array row specified by register *rd*. This instruction is similar to `mtga` except that the row number and the R field of `mtga` are given by the value of register *rd* as follows:

32 … 11	1	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	row number	R

As with `mtga`, if the least significant bit of *rd* (bit R) is 0, the destination of the copy is the concatenation of the sixteen Z registers in columns 4 through 19 of the specified row; whereas if the least significant bit of *rd* is 1, the destination is the concatenation of the sixteen D registers in columns 4 through 19. Column 4 receives the least significant 2 bits of the value copied and column 19 receives the most significant 2 bits.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the copy is performed.

Copy word from array, variable row

32 … 25	21	16	11	5	0
0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 0 1 0	0 0 0 0 0

`mfgav` *rt*,*rd*

This instruction is identical to `mtgav` except that the direction of the copy is reversed.

Figure 38(c): Instructions for copying a word to/from a variable array row.

Copy word to array, variable row, low columns

32 — 25	25 — 21	21 — 16	16 — 11	11 — 5	5 — 0
0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 1 0 1	0 0 0 0 0

`mtgavy` *rt*,*rd*

This instruction is identical to `mtgav` except that the destination of the copy is columns 0 through 15 of the specified row. Column 0 receives the least significant 2 bits of the value copied and column 15 receives the most significant 2 bits.

Copy word from array, variable row, low columns

32 — 25	25 — 21	21 — 16	16 — 11	11 — 5	5 — 0
0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 1 0 0	0 0 0 0 0

`mfgavy` *rt*,*rd*

This instruction is identical to `mtgavy` except that the direction of the copy is reversed.

Copy word to array, variable row, high columns

32 — 25	25 — 21	21 — 16	16 — 11	11 — 5	5 — 0
0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 0 0 1	0 0 0 0 0

`mtgavz` *rt*,*rd*

This instruction is identical to `mtgav` except that the destination of the copy is columns 16 through 22 of the specified row. Only the least significant 14 bits of source register *rt* are copied. The most significant 18 bits of *rt* are ignored. Column 16 receives the least significant 2 bits of the value copied and column 22 receives the most significant 2 bits.

Copy word from array, variable row, high columns

32 — 25	25 — 21	21 — 16	16 — 11	11 — 5	5 — 0
0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 0 0 0	0 0 0 0 0

`mfgavz` *rt*,*rd*

This instruction is identical to `mtgavz` except that the direction of the copy is reversed. The most significant 18 bits of destination register *rt* are zeroed.

Figure 38(d): Instructions for copying a word to/from the leftmost or rightmost columns of an array row.

Copy 2 words to array

32 25 21 16 11 5 0

0 1 0 0 1 1 1	1 0 1 0	rt	rd	match code	clock count

`mtga2` *rt*,*rd*,*match*,*count*
`mtga2` *rt*,*rd*,*match*

Copies the values in registers *rt* and *rd* to the middle 16 logic blocks of some set of array rows. The destination rows are determined indirectly through a 6-bit *match code* encoded in the instruction. The values of registers *rt* and *rd* are placed on memory buses 0 and 2 respectively, and each row does or does not latch one of these values according the current configuration's response to the given match code. The control block on each row can be configured to respond to a specific match code value by latching from either bus into either the Z or D registers in columns 4 through 19 of that row. (Additional information can be found in Section 3.2.1.)

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. The copy is then performed, after which the clock counter is set to the 5-bit unsigned integer constant encoded in the instruction.

The *match* and *count* arguments must be integer constants. If *count* is not given it defaults to zero.

Figure 38(e): Instruction for copying two words to two or more array rows.

Branch on array condition true

32 16 0

0 1 0 0 1 1 0 1 0 0 0 0 0 0 0 1	offset

`bgat` *offset*

Causes instruction execution to branch if the array condition flag is set (*true*). Like other MIPS branch instructions, the instruction following this one is executed before any branch is taken. The instruction following this one must not be a branch.

Branch on array condition false

32 16 0

0 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0	offset

`bgaf` *offset*

Causes instruction execution to branch if the array condition flag is clear (*false*). Like other MIPS branch instructions, the instruction following this one is executed before any branch is taken. The instruction following this one must not be a branch.

Figure 38(f): Instructions for branching on the state of the array condition flag.

Load array configuration

32 25	21	16	11	5	0
0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	1 1 0 1 1 0	0 0 0 0 0

`gaconf` *rt*

Loads a configuration from memory and makes it active. Register *rt* gives the starting address of the configuration in memory.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. Any array reads still in progress are then cancelled, and the existing array allocation, if any, is released. Sufficient space is allocated within the array to hold the specified configuration, and the configuration is loaded and made active. The Z and D registers in the newly allocated space are zeroed. This instruction is equivalent to the sequence of a `gaalloc` instruction followed by `gaconfo`.

The copy of the configuration in memory must not change until a flush configuration instruction (`gacinv`) is executed for this address.

Reset array

32 25	21	16	11	5	0
0 1 0 0 1 1 1	0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 1 0 0 1 0	0 0 0 0 0

`gareset`

Resets the array, releasing the existing array allocation.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. Any array reads still in progress are then cancelled, and the existing array allocation, if any, is released.

Flush array configuration from cache

32 25	21	16	11	5	0
0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	0 1 0 0 0 0	0 0 0 0 0

`gacinv` *rt*

Flushes the configuration or array allocation at the address given by *rt* from the configuration cache.

Figure 38(g): Instructions for loading and managing array configurations.

Allocate array space

32	25	21	16	11	5	0
0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	1 1 0 0 1 0	0 0 0 0 0	

`gaalloc` *rt*

Allocates space within the array for a configuration, without actually loading a configuration. Register *rt* gives the address of a word in memory specifying the number of rows to allocate.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. Any array reads still in progress are then cancelled, and the existing array allocation, if any, is released. The new allocation is put into effect, with all array rows inactive. The Z and D registers in all of the newly allocated space are zeroed.

The memory location pointed to by *rt* must not change until a flush configuration instruction (`gacinv`) is executed for this address.

Load array configuration overlay

32	25	21	16	11	5	0
0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 1 0 1 0 0	clock count	

`gaconfo` *rt*,*rd*,*count*
`gaconfo` *rt*,*rd*

Loads a configuration from memory into the previously allocated array space, while preserving array data state. Register *rt* gives the starting address of the configuration in memory, and register *rd* gives the first allocated row at which to load the overlaying configuration.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. The specified configuration is then loaded and made active, after which the array clock counter is set to the 5-bit unsigned integer constant encoded in the instruction.

The configuration being loaded cannot extend outside the current allocated array space. Although the existing array allocation remains in effect in its entirity, *only* the rows of the overlaying configuration are made active. The contents of the Z and D registers within the allocated array space are unaffected by this operation.

The copy of the configuration in memory must not change until a flush configuration instruction (`gacinv`) is executed for this address.

Figure 38(h): Instructions supporting configuration overlays.

Load array queue control

32 25 21 16 11 5 0

0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 1 0 0 0	0 0 0 0 0

`galqc` *rt*,*rd*

Loads the control registers for the queue specified by register *rd* with the 20 bytes at the address given by register *rt*. The value of register *rd* must be an integer in the range of 0 to 2 inclusive, indicating one of the three array memory queues. Details about the memory queues and the queue control registers can be found in Section 3.3 and Figure 51.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the load is performed.

Store array queue control

32 25 21 16 11 5 0

0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 1 0 0 1	0 0 0 0 0

`gasqc` *rt*,*rd*

Stores the control registers for the queue specified by register *rd* into 20 bytes starting at the address given by register *rt*. The value of register *rd* must be an integer in the range of 0 to 2 inclusive, indicating one of the three array memory queues.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the store is performed.

Figure 38(i): Instructions for loading and storing the control registers of an array memory queue.

Stop array

32	25	21	16	11	5	0
0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0	

`gastop` *rt*
`gastop`

Zeros the clock counter, halting array execution. Register *rt* gets the value that the counter had before being zeroed.

The assembler allows the destination operand to be dropped, in which case *rt* is set to the MIPS pseudo-register `$0` (zero register).

Copy word from array control register

32	21	16	11	0
0 1 0 0 1 1 0 0 0 1 0	rt	zd	0 0 0 0 0 0 0 0 0 0 0	

`cfga` *rt*,*zd*

Copies the array control register *zd* to processor register *rt*. The 5-bit *zd* field must be one of the following integers:

0 The version register, which has the format

32	16	8	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	implementation	revision	

1 The number of bytes that `gasave` writes to memory. (This is constant for a given implementation/revision.)

3 The pointer that was the argument to `gaalloc` or `gaconf` when the current array allocation was made.

4 The pointer to the configuration in memory that was the argument to `gaconfo` or `gaconf`.

5 The row offset that was the argument to `gaconfo`, or zero if the active configuration was loaded by `gaconf`.

The assembler accepts for *zd* either the notation `$`*n* or `$gacr`*n*.

Figure 38(j): Instructions for stopping the array and retrieving array state.

Save internal array state

32 25 21 16 11 5 0

0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	1 1 1 0 0 1	0 0 0 0 0

`gasave` *rt*

Saves the internal state of the array to memory at the address given by *rt*. The internal state includes in particular the status of pending reads from memory. The amount of memory needed to store the saved state can be discovered by reading the `$gacr1` control register using the `cfga` instruction.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the store is performed.

Restore internal array state

32 25 21 16 11 5 0

0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	1 1 1 0 0 0	0 0 0 0 0

`garestore` *rt*

Loads the internal state of the array from memory at the address given by *rt*. This instruction also stalls long enough to ensure that conbinatorial signals in the array have been allowed to settle, assuming a maximum path delay of 8 array clock cycles.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the load is performed.

Figure 38(k): Instructions for saving and restoring internal array state.

Instruction	Bit positions	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
cfga	32, 21, 16, 11, 0	0 1 0 0 1 1 0 0 0 1 0	rt	zd	0 0 0 0 0 0 0 0 0 0 0		
bgaf	32, 16, 0	0 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0	offset				
bgat	32, 16, 0	0 1 0 0 1 1 0 1 0 0 0 0 0 0 0 1	offset				
gastop	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0
gabump	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	0 0 0 0 0	rd	0 0 0 0 1 0	0 0 0 0 0
gacinv	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	0 1 0 0 0 0	0 0 0 0 0
mfgavz	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 0 0 0	0 0 0 0 0
mtgavz	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 0 0 1	0 0 0 0 0
mfgav	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 0 1 0	0 0 0 0 0
mtgav	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 0 1 1	0 0 0 0 0
mfgavy	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 1 0 0	0 0 0 0 0
mtgavy	32, 25, 21, 16, 11, 5, 0	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 0 1 0 1	0 0 0 0 0

Figure 39: List of added instructions in encoding order.

	32–25	25–21	21–16	16–11	11–5	5–0
galqc	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 1 0 0 0	0 0 0 0 0
gasqc	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 0 1 0 0 1	0 0 0 0 0
gareset	0 1 0 0 1 1 1	0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 1 0 0 1 0	0 0 0 0 0
gaalloc	0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	1 1 0 0 1 0	0 0 0 0 0
gaconfo	0 1 0 0 1 1 1	0 0 0 0	rt	rd	1 1 0 1 0 0	clock count
gaconf	0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	1 1 0 1 1 0	0 0 0 0 0
garestore	0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	1 1 1 0 0 0	0 0 0 0 0
gasave	0 1 0 0 1 1 1	0 0 0 0	rt	0 0 0 0 0	1 1 1 0 0 1	0 0 0 0 0
mfga	0 1 0 0 1 1 1	1 0 0 0	rt	row number	R	clock count
mtga	0 1 0 0 1 1 1	1 0 0 1	rt	row number	R	clock count
mtga2	0 1 0 0 1 1 1	1 0 1 0	rt	rd	match code	clock count

Figure 39 continued.

3.2 Array control blocks

The control blocks at the left end of each row help interface between the array on the one hand and the processor and memory on the other. The functions a control block can perform include:

- zero the clock counter (thus halting array execution);
- interrupt the processor;
- set the array condition flag (recall Section 3.1.3);
- latch a value being transfered by the `mtga2` instruction into the Z or D registers of the row (Section 3.1.2);
- initiate a memory access at an arbitrary address;
- initiate a read or write through an array memory queue.

As always, the active configuration determines which if any of these functions each control block might perform. Figure 40 shows the general encoding of the configuration for a control block. Like a logic block, a control block's configuration is 64 bits, with four 2-bit inputs, A, B, C, and D, taken from adjacent wires. These inputs are used to control some subset of the functions listed above, according to the *mode* in which the control block is configured.

Regardless of mode, the 8 bits of input are always reduced down to three control signals as illustrated in Figures 41 and 42. First, each individual 2-bit input is reduced to a single bit, either by discarding one of the bits or by logically *or*-ing the two bits (Figure 42). The resulting A' signal is then used to gate each of the corresponding B', C', and D' signals to construct the three control signals. The three control signals are thus generated directly from the B, C, and D inputs, except that A acts as an enable for all three signals.

Any of the four inputs can be fixed to binary constant 00 or 10, the same as for a logic block. Otherwise, a control block input can only come from a local horizontal wire (H wire), either from above or below the control block. Control blocks have no outputs, so there are no vertical wires associated with the column of control blocks. Aside from the fewer options, the encoding of control block inputs is identical to that for logic blocks (Figure 11).

For timing purposes, inputs that are not constant must come directly from a logic block register across the connecting H wire to the control block (Figure 43). A logic block register that supplies an input to a control block in this way is known as the *upstream register* for that control input.

3.2.1 Processor interface blocks

In processor interface mode, a control block can perform various functions connected with the main processor. The format of a processor interface configuration is given by Figure 44. The simplest functions involve the three control signals, which can manipulate the array condition flag value, zero the clock counter, and cause the main processor to take an interrupt. Processor interface mode also provides the means by which the control block's row can latch one of the two words copied from the processor register file by the `mtga2` instruction.

The B' input to the control block can be used to determine the value of the array condition flag. If A' and B' are both 1, the condition flag is set (*true*); otherwise, the value of the condition flag depends on the other control blocks configured in processor interface mode. If the array condition flag is not set by any control block, the flag is clear (*false*). The condition flag is not sticky, so it can transition

64	58	56	50	48	42	40	34	32
A in	A′	B in	B′	C in	C′	D in	D′	

32	5	3	0
mode-specific fields	Hdir	mode	

[63..58] A in	
000000	$A = 00$ (binary)
000001	$A = 10$ (binary)
100010	$A =$ leftmost H wire pair above
⋮	⋮
101010	$A =$ rightmost H wire pair above
110010	$A =$ leftmost H wire pair below
⋮	⋮
111010	$A =$ rightmost H wire pair below

[57..56] A′	
00	$A' = A_0$
10	$A' = A_1 \vee A_0$
11	$A' = A_1$

[4..3] Hdir	
00	H wires driven from right end (shift left)
01	H wires driven from center
10	H wires driven from left end (shift right)

[2..0] mode	
000	no function
010	processor interface
110	memory interface

Figure 40: Control block configuration encoding.

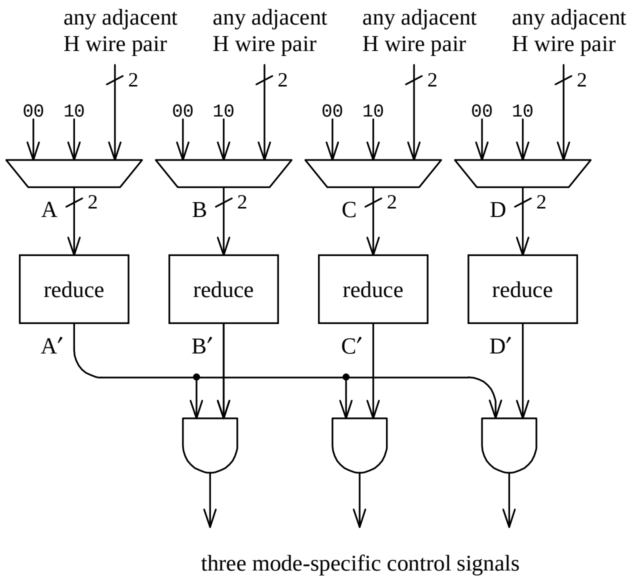


Figure 41: Control block signals.

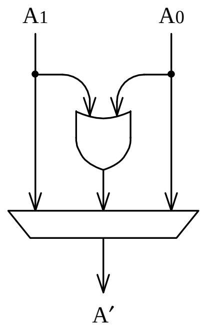


Figure 42: The reduction functions.

from clear to set to clear again as determined each clock cycle by the corresponding control signals.

The C' input allows the array to zero the clock counter, and the D' input makes it possible for the array to interrupt the processor. If the A' and C' inputs are both 1, the array clock counter is zeroed at the end of the current array clock cycle, thus halting array execution. If A' and D' are both 1, the main processor is forced to take an interrupt. Note that array execution is not directly affected by any processor interrupts.

Independent of these three control functions, other configuration fields control the latching of words copied from the processor with the `mtga2` instruction (Section 3.1.2). Recall that the `mtga2` instruction makes its two operands visible on two of the memory buses running through the array. If the match code specified in the control block's configuration matches that in the `mtga2` instruction, one

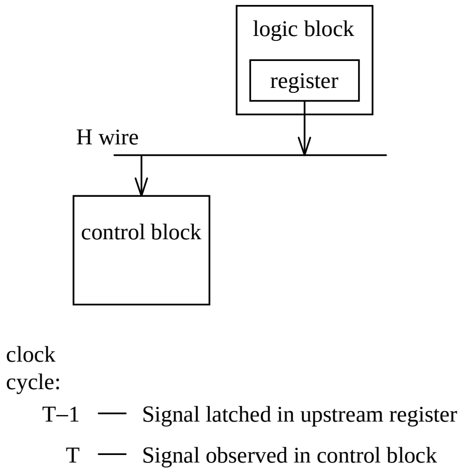


Figure 43: Valid connection to a control block. A control block input must come directly over a local horizontal wire from a logic block register in the same row or the row above.

64 58 56 50 48 42 40 34 32

A in	A′	B in	B′	C in	C′	D in	D′

32 22 16 14 13 10 8 5 3 0

0 0 0 0 0 0 0 0 0 0	match code	size_r	R	0 0 0	bus	0 0 0	Hdir	0 1 0

A' $\rightarrow$ enable
B' $\rightarrow$ condition flag
C' $\rightarrow$ zero clock counter
D' $\rightarrow$ interrupt processor

[15..14] size_r	
00	8 bits
01	16 bits
10	32 bits

[13] R	
0	load Z registers
1	load D registers

[9..8] bus	
00	`mtga2` *rt* operand
10	`mtga2` *rd* operand

Figure 44: Configuration for a control block in processor interface mode.

64 58 56 50 48 42 40 34 32

A in	A′	B in	B′	C in	C′	D in	D′

32 30 27 24 22 21 18 16 14 13 10 8 5 3 0

type	0 0 0	delay	size_a	N	0 0 0	K	size_r	R	0 0 0	bus	0 0 0	Hdir	1 1 0

Figure 45: Configuration for a control block in memory interface mode. Details about the various fields are covered in Figures 47, 48, and 52.

of these two words will be latched into registers along the control block's row. Three fields of the configuration (Figure 44) determine: (1) which word is latched (the *rt* or *rd* operand to `mtga2`), (2) which registers will be loaded (Z or D), and (3) the number of bits to load (8, 16, or 32). If 32 bits are to be loaded, the chosen value will be latched into the Z or D registers in columns 4 through 19 of the row. If only 16 bits are to be loaded, only columns 4 through 11 are affected, and the registers in columns 12 through 19 are left untouched. Likewise, if only 8 bits are to be loaded, only registers in columns 4 through 7 are affected by the operation.

The `mtga2` instruction can only execute when the array is halted, so the instruction cannot interfere with operation of the array. Unlike the other instructions for transfering data to the array, any number of rows can load either of the two values copied by `mtga2`.

3.2.2 Memory interface blocks

Memory accesses can be initiated from the reconfigurable array without direct processor intervention. A memory access proceeds in two steps: the *initiate step* starts the access by providing a memory address, and the *transfer step* transfers the data (Figure 46). The address is read from the Z registers of a selected row, over a special *address bus* that runs parallel to the four memory buses already mentioned. Up to four contiguous words can be read or written in one memory access, where the word size is selectable as either 8, 16, or 32 bits. Each word is transferred over a separate memory bus.

For memory writes, the initiate and transfer steps must occur together in the same clock cycle. For reads, the initiate step necessarily precedes the transfer step. Only one demand access to memory can be initiated in each array clock cycle, although multiple memory accesses may be in different stages of progress at any one time.

The array sees the same memory hierarchy as the main processor, including all data caches. Misses in the first level data cache may 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 data cache. Array memory accesses may also generate page fault traps as discussed later.

In *memory interface* mode, a control block has the ability either to initiate a memory access or to participate in the transfer of data, or both. Figure 45 shows the format of the memory interface configuration. Because the initiate and transfer phases are controlled independently, the configuration fields associated with the initiate step will be presented first, separately from those concerned with the transfer step.

Figure 47 highlights the parts of a memory interface configuration that control the initiation of memory accesses. The actual instigation of a memory access is controlled by the B' signal, while the direction of access (read versus write) is determined by D'. A demand access to memory is initiated whenever A' and B' are both 1. If D' is 0 at that time, the access will be a read; otherwise it will be either a write or a prefetch, depending on the configuration. When a control block initiates a demand memory access, the contents of the Z registers in the logic blocks in columns 4 through 19 of the same row are sent over the address bus to provide a 32-bit address for the memory system.

Other configuration fields control various apects of the memory access. The size_a and K fields choose the word size and number of words to access, respectively. The largest possible access is to four contiguous 32-bit words, while the smallest is to a single 8-bit "word." The word size and the number of words must

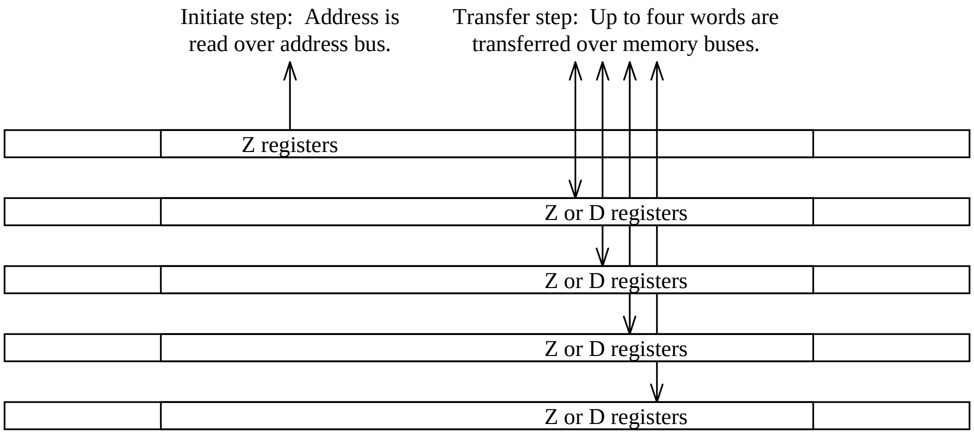


Figure 46: The two steps of a memory access initiated by the array.

each be a power of two within these ranges.

If an access is not a read (that is, if $D' = 1$), it is either a write or a prefetch. The type field of the configuration (Figure 47) determines whether a non-read memory access is a write or a prefetch, and also whether a cache miss should cause data to be brought into the cache. Since prefetches are performed solely for the purpose of bringing data into the cache, it makes no sense not to do cache allocation on misses in this case. Normal reads and writes may be configured for cache allocation on misses or not.

When the access word size is larger than a byte, the given address may not be aligned on a natural word-size boundary. The configuration chooses one of two possibilities: either the least significant bits of the address are ignored, or a nonaligned memory access is performed at the specified address. The number of bits ignored is dependent on the word size: 1 bit if the word size is 16 bits, and 2 bits if the word size is 32 bits.

Finally, the delay field in the configuration determines the perceived delay for read accesses. This field is ignored for writes and prefetches. The timing details of memory accesses are covered later in this section.

Although any number of control blocks can be configured as memory interfaces, only one control block can initiate a memory access during any array clock cycle.

The transfer step performs the actual movement of data, either simultaneously with the initiate step in the case of writes, or after the data has been read from memory. Figure 48 shows the memory interface configuration fields associated with the transfer step. The transfer of data into or out of the array on a memory access is similar to the latching of data transferred from the processor by the `mtga2` instruction (Section 3.2.1). Instead of a match code, the C' input to the control block decides, for each clock cycle, whether a transfer into or out of the row occurs on that cycle. The D' signal indicates the direction of transfer, the same as it does for the initiate step.

Each word of the as many as four words transferred has a memory bus dedicated to it during the transfer. The first word at the given memory address is copied over bus 0; and if the access involves more than one word, sub-

64–58	58–56	56–50	50–48	48–40	40–34	34–32
A in	A′	B in	B′		D in	D′

32–30	30–27	27–24	24–22	22–21	21–18	18–16	16–3	3–0
type	0 0 0	delay	$\mathrm{size_a}$	N	0 0 0	K		1 1 0

A' → enable
B' → initiate memory access
D' → 0 = read, 1 = write or prefetch

[31..30] type	
01	demand access, read/prefetch, cache allocate
10	demand access, read/write, cache allocate
11	demand access, read/write, no cache allocate

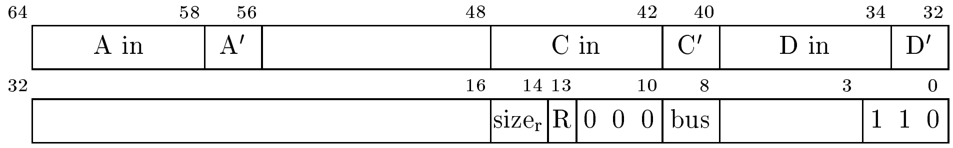
[26..24] delay	
000	1 cycle
⋮	⋮
111	8 cycles

[23..22] $\mathrm{size_a}$	
00	8 bits
01	16 bits
10	32 bits

[21] N	
0	aligned address (ignore bottom bits)
1	possibly nonaligned address

[17..16] K	
00	demand access 1 word
01	demand access 2 words
10	demand access 4 words

Figure 47: Memory interface configuration fields associated with the initiate step of a demand memory access.



A' → enable
C' → load/store registers over bus
D' → 0 = load, 1 = store

[15..14] size_r	
00	8 bits
01	16 bits
10	32 bits

[13] R	
0	load/store Z registers
1	load/store D registers

[9..8] bus	
00	bus 0
01	bus 1
10	bus 2
11	bus 3

Figure 48: Memory interface configuration fields associated with the transfer step of a memory access.

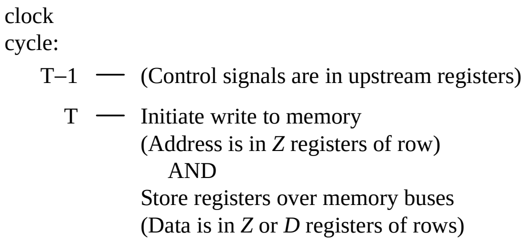


Figure 49: Timing of a memory write executed by the array. The write "occurs" in the active cycle following the control signals being applied (see text).

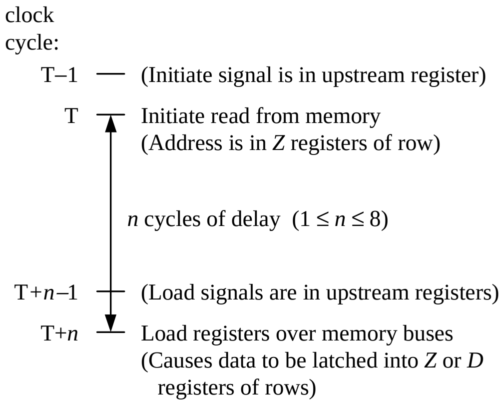


Figure 50: Timing of a memory read executed by the array.

sequent words are copied over buses 1, 2, and 3, in that order. For example, a memory access of two words involves buses 0 and 1: bus 0 for the word at the given address in memory, and bus 1 for the next contiguous word in memory.

Following the initiation of a memory read of n words, and after a specific number of array clock cycles have elapsed (discussed below), buses 0 through $n-1$ will contain the values read from memory. At that time, the control blocks on the rows into which these values should be latched must signal the transfer step.

For writes, the initiate and transfer steps are signalled simultaneously. Exactly one word must be driven onto each of the n buses. Figure 49 shows the timing for a write.

Conceptually, the write occurs in the clock cycle following the control signals being applied. If the clock counter is also zeroed in clock cycle T, the write will not occur until the clock counter is subsequently given a nonzero value, continuing execution of the same configuration. If the current configuration is never resumed, the write will never actually occur, despite having been initiated.

Figure 50 shows the timing for a read. For reads, the delay field of the memory interface configuration specifies the number of array clock cycles by which the data will be delayed. If this is at least as great as the actual latency of a read operation, execution of the array will not be stalled waiting for the read to return. Otherwise, an implementation must stall the array sufficiently to give the appearence that the data was returned in the specified number of array clock cycles.

3.3 Array memory queues

Besides being able to request memory accesses directly, the array has available to it three *memory queues* that can increase the performance of sequential accesses. The array reads or writes to/from a queue much as it does directly to/from memory, except that it does not supply an address or other information about the access. A memory queue is programmed by the main processor with this information in advance, using the `galqc` instruction. Figure 51 shows the format of the 20 bytes of control information that are loaded from memory into a memory queue's controller by the `galqc` instruction. A corresponding `gasqc` instruction writes back this information to memory in the same format to facilitate context switches.

Like demand memory accesses, a memory queue can be programmed to transfer up to four words on each request. Unlike a demand access, the four words can be matched to buses arbitrarily, so that the first word is not necessarily transferred over bus 0, etc. The last four bytes loaded into the queue controller assign a bus to each word (Figure 51).

The configuration of a control block that allows it to initiate a queue access is a variation of the one for demand memory accesses, as seen in Figure 52. As before, an access is initiated whenever A' and B' are both 1. The only additional information encoded in the configuration is the queue number to access. The direction of transfer (read or write) is determined by the queue itself and is not decided by the D input to the control block as it is for demand accesses. The address bus is not used for queue accesses. The transfer step of a queue access is identical to that for a demand access, keeping in mind that the association between buses and words is not fixed but is set by the queue controller.

Ultimately, a queue access has the same affect as a demand memory access at the ad-

32	25 24		17 16		9 8	0
0 0 0 0 0 0 0	E	0 0 0 0 0 0 0	D	0 0 0 0 0 0 0	A	0 0 0 0 0 0 0 0

32	26 24		18 16	0
0 0 0 0 0 0	size	0 0 0 0 0 0	K	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

32 0
address

32 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

32	26 24		18 16		10 8		2 0
0 0 0 0 0 0	bus_0	0 0 0 0 0 0	bus_1	0 0 0 0 0 0	bus_2	0 0 0 0 0 0	bus_3

E	
0	queue disabled
1	queue enabled

D	
0	read
1	write

A	
0	no cache allocate
1	cache allocate

size	
00	8 bits
01	16 bits
10	32 bits

K	
00	1 word per access
01	2 words per access
10	4 words per access

bus_n	
00	word n on bus 0
01	word n on bus 1
10	word n on bus 2
11	word n on bus 3

Figure 51: Format of a queue control record.

64	58	56	50	48	32
A in	A′	B in	B′		

32	30	18	16		3	0
type	0 0 0 0 0 0 0 0 0 0 0 0	Q			1 1 0	

$A' \rightarrow$ enable
$B' \rightarrow$ initiate queue access

[31..30] type	
00	queue access

[17..16] Q	
00	queue 0
01	queue 1
10	queue 2

Figure 52: Memory interface configuration fields associated with the initiate step of a memory queue access. (Compare with Figure 47.)

clock
cycle:
T–1 — (Control signals are in upstream registers)
T — Initiate write to memory queue
AND
Store registers over memory buses
(Data is in *Z* or *D* registers of rows)

Figure 53: Timing of a write to an array memory queue. The write "occurs" in the active cycle following the control signals being applied (see text).

clock
cycle:
T–1 — (Initiate signal is in upstream register)
T — Initiate read from memory queue
(Load signals are in upstream registers)
T+1 — Load registers over memory buses
(Causes data to be latched into *Z* or *D* registers of rows)

Figure 54: Timing of a read from an array memory queue. This is equivalent to a demand memory read with a fixed delay of 1 clock cycle.

dress stored within the queue controller (recall Figure 51). After each access to a memory queue, the stored address is incremented to the next contiguous byte following the last one read or written. A `gasqc` instruction writes out a queue control record with this updated address, so that `galqc` can properly restore the state that the queue had at the time `gasqc` was executed.

Figures 53 and 54 show the timing of a queue write and a queue read, respectively. The timing for a queue write is indistinguishable from that of a demand memory write, while a queue read appears the same as a demand read with the delay fixed at 1 clock cycle. Like a demand memory write, a queue write is not committed until one array clock cycle following the initiation of the write by the control block. If the clock counter becomes zero in the same cycle that the queue write is initiated and if array execution is never resumed, the write will not occur.

Each of the three queues can be accessed on every array clock cycle, simultaneously with the initiation of a new demand memory access every cycle. This makes it possible to achieve four independent memory accesses per clock cycle to/from the array. However, each memory bus can transfer only one word during any given cycle, so in total a maximum of four 32-bit words can be accessed each cycle.