A Reconfigurable Arithmetic Array for Multimedia Applications

Alan Marshall, Tony Stansfield, Igor Kostarnov
Hewlett Packard Laboratories
Filton Road,
Bristol BS34 8QZ, UK
+44 (0)117 922 8207 | 9841 | 8197

alanm | ais | iak@hpl.hp.com

Jean Vuillemin
Ecole Normale Supérieure
45 rue d'Ulm
75230 Paris Cedex 5, France
+33 (1) 4432 2074

Jean.Vuillemin@ens.fr

Brad Hutchings
Brigham Young University
459 Clyde Building
Provo, Utah 84602
+1 (801) 378-2667

hutch@ee.byu.edu

ABSTRACT

In this paper we describe a reconfigurable architecture optimised for media processing, and based on 4-bit ALUs and interconnect.

Keywords

Reconfigurable computing, multimedia, 4-bit ALU, FPGA.

1. INTRODUCTION

The high computational workloads in multimedia applications have motivated a number of styles of acceleration. These include intense kernel codes, specialised instructions, specific multimedia processors, custom hardware add-ons and *reconfigurable computing*. Such accelerators are implemented as independent processors, co-processors or IP components in ASICs.

Experimental work on reconfigurable computing has focussed on FPGAs as the only available implementation technology. While many successful systems have been built from single-bit output FPGA logic cells, there appear to be limits to this approach when compared to ASICs: low arithmetic density, reduced clock speed and low internal RAM density and bandwidth, as well as increasingly higher reconfiguration times.

In light of this experience, HP Labs has been developing a *reconfigurable arithmetic array* (RAA), termed CHESS and aimed as a component of an ASIC or processor datapath. It is provides high computational density, and sufficient distributed register and memory resource for important multimedia algorithm cores. CHESS also offers software flexibility, strong scalability and advanced features for dynamic reconfiguration.

This paper describes the goals of this work, outlines the architecture and implementation, presents examples of use and discusses results. Software development toolchains and system contexts for such an architecture are not discussed in this paper.

2. RECONFIGURABLE LOGIC SHORTCOMINGS FOR ARITHMETIC

Reconfigurable computing has been an active research area for nearly 10 years. Most previous work has been based on commercially available FPGAs [16]. This has demonstrated that reconfigurable computing can be effective for high-end systems [2], with experiments migrating from multiple FPGA solutions to the use of a single large FPGA. But so far the technology has not been adopted for mass-market products. Discussion with system designers indicates that such technologies are too expensive in terms of used silicon area when supporting arithmetic for application cores of importance.

For *arithmetic data flow* applications, bit-level FPGA implementations duplicate many resources to support high-level operations, such as wide arithmetic operations or routing multi-bit buses. In many cases the bulk of the active silicon is engaged in such emulation of a 'wider' machine. A number of commercial architectures [12, 14, 16] cluster functional blocks, allowing them to support fast carry structures for arithmetic. But these architectures do not exploit the clustering to reduce the configuration memory overhead, presumably because doing so would compromise their ability to support general-purpose logic. A number of academic architectures [3, 4, 5, 6, 7, 8, 11] (discussed in Section 7) have recognised this issue and concentrate on denser support for arithmetic applications even at the expense of generality. These architectures are based on one or both of two major techniques: the sharing of configuration bits between multiple bits of word width and the use of function blocks tuned for arithmetic applications.

Another shortcoming is the low memory density of current FPGAs: only small on-chip memories are practical, so external SRAM must be used for many multimedia applications with consequent impact on available memory bandwidth. Long reconfiguration times, measured in tens of ms, also limit the value of reconfigurability in situations such as video processing.

3. ARCHITECTURAL CHOICES FOR CHESS

Our principal goals for CHESS were to increase both arithmetic computational density and the bandwidth and capacity of internal memories significantly beyond the capabilities of current FPGAs, whilst enhancing flexibility. Dense implementation of bit-level logic was not a principal goal for us. The choices we made were:

4-bit ALUs
The fundamental computational unit is a 4-bit ALU, with a primary set of 16 instructions. This provides efficient arithmetic capabilities, suitable for cascading to useful media widths, or for supporting nibble-serial implementation styles. ALU instructions can be either constant or dynamic. Constant instructions are stored as a part of the configuration. Dynamic instructions are generated via user circuitry that is connected to the instruction input of the ALU.

4-bit bus wiring
The entire user-visible routing structure is based on 4-bit buses, supporting efficient transmission of user data or ALU instructions.

Switchboxes
Each ALU is paired with an adjacent switchbox that can operate in two modes. In its default mode the switchbox serves as a crosspoint switch with 64 connections, used to connect horizontal and vertical buses that physically pass over the switchbox. In the second mode the 64 bits of configuration memory that configure the crosspoint switches are reclaimed and used to implement a 16W*4b RAM.

Chessboard layout
The ALUs and routing switchboxes are laid out in a two-dimensional symmetric chessboard. This supports strong local connectivity between ALUs and gives an effective routing network using only 50% of the array area, much less than is typical for FPGAs.

Embedded block RAMs
The CHESS architecture allows block RAMs to be inserted throughout the basic array of ALUs and switchboxes. Our baseline design provides one 256W*8b RAM per 16 ALUs.

Speed and hierarchical line lengths
Clock speed factors directly into computational density, and so it is important to prevent long connections from limiting the clock speed severely. To address this issue, CHESS provides two register/buffers per switchbox (in addition to the register in each ALU) to allow heavy pipelining of long connections. Buses are provided at power-of-two lengths, with the longer buses having only five connections, directly to the retiming registers/buffers. This allows long connections to be made without limiting clock speed.

Small configuration memories
Our choices of 4-bit wiring and 4-bit ALUs allow CHESS to have about 100 configuration bits for each ALU and its associated wiring. The small number of configuration bits is a major contributor to the density of CHESS, and also allows rapid reconfiguration. A 512 ALU CHESS array could be completely reconfigured in 40μs over a standard 32bit 20ns burst-mode DRAM interface for example, and partially reconfigured in proportionally less time.

No run-time reconfiguration
As CHESS has rapid off-line reconfiguration, and also the ability to feed instructions into the ALUs for cycle-by-cycle changes in functionality at run time, we saw no need to allow part of CHESS to be reconfigured while another part is operating. This decision simplified many design issues and avoided the significant circuit and CAD tool overheads of avoiding transient drive fights.

4. APPLICATION BENEFITS OF CHESS

Computational density
The computational density of CHESS allows many of our target applications to be mapped to one configuration of an array that is small enough to be integrated into a single chip system. It is important to reach this threshold for several reasons:

- It brings the benefits of delayed design commitment and in-system reconfiguration without the cost and pin restrictions of separately packaged reconfigurable logic.
- It allows multiple wide interfaces to other on-chip circuitry. In particular, many applications can benefit from multiple large on-chip RAMs with wide interfaces to the datapaths.
- Fitting an application into one CHESS configuration reduces the use made of run-time reconfiguration, cutting the associated overheads of reconfiguration time and use of memory bandwidth.

Memory bandwidth
Many of our image processing applications require memory of modest size but high bandwidth for line buffers and translation tables. Distributed on-chip memories can significantly reduce off-chip memory traffic, which would otherwise limit system performance in many cases.

Flexibility
By providing the ability to convert switchboxes into 16W*4b RAMs where needed by an application, CHESS allows RAM to be traded for computation in a flexible way. This approach gives higher memory density than converting the few ALU configuration bits into memory (the conventional technique). Being able to build small memories as needed within the array helps keep memory references to frequently used small data structures on-chip, close to their use, and permits multiple parallel accesses in each clock cycle. These 16W*4b memories can also be used as 4-input, 4-output LUTs for operations that do not map well onto ALU instructions. This feature can also be used to make arbitrary interconnections at the bit level, which are not supported by CHESS' 4-bit wiring.

Flexibility in another dimension comes from the dynamic ALU instructions, which allow specialised processors to be built within CHESS. By allowing complex functions to be cast into programs for these processors, this technique greatly increases the range of applications that can be supported within CHESS.

Rapid reconfiguration
One of the main benefits of rapid reconfiguration is in allowing multiple processing phases to be applied to a stream of data with minimal buffering. If processing needs to be halted during

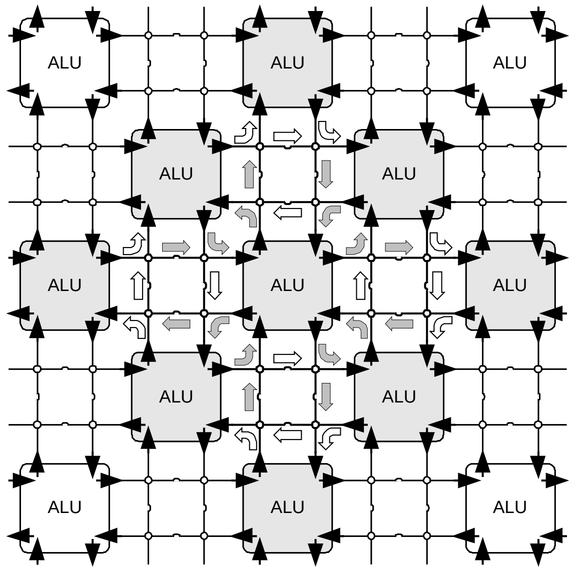

Figure 1 CHESS layout and nearest neighbour wiring

reconfiguration, then the amount of buffering needed on the data stream is directly related to the reconfiguration time. So for example, a 512 ALU CHESS could handle three successive processing steps on each frame of a 50fps video stream, being completely reconfigured for each phase every 1/16th of a frame, for a 10% configuration time overhead. This would require buffering for just 1/16th of a frame. If the changes between processing steps can be achieved with partial reconfiguration or with the use of dynamic ALU instructions, then the overheads would be still lower and the reconfiguration rate might be increased.

5. THE CHESS ARCHITECTURE IN DETAIL

Our major choice was in the granularity of the data unit. We chose 4-bit nibbles. This size is just large enough to support a useful 'instruction' set for an ALU, whilst providing efficient arithmetic for the 8/12/16bit data sizes prevalent in multimedia. 8-bit solutions such as MATRIX [7] are also feasible, but we believe that they would tend to have more complex function units than CHESS to balance the use of transistors and metal tracks.

CHESS' functional units are 4-bit ALUs, and the connections are 4-bit buses. This reduces the number of bits of configuration memory by nearly a factor of four compared to a conventional FPGA, allowing increased density. The smaller configuration memory also allows faster reconfiguration.

CHESS' chessboard style (see Figure 1) contrasts strongly with most of the other arithmetic-oriented reconfigurable architectures using a one-dimensional bus-centred approach. Each ALU is adjacent to four switchboxes, and each switchbox is adjacent to four ALUs. This allows very powerful local connectivity, with each ALU having input and output buses on all four sides, and being able to send data to or receive data from any of the eight surrounding ALUs as shown in Figure 1 by the grey arrows. In regular datapath applications, good placement allows these local connections to be exploited heavily.

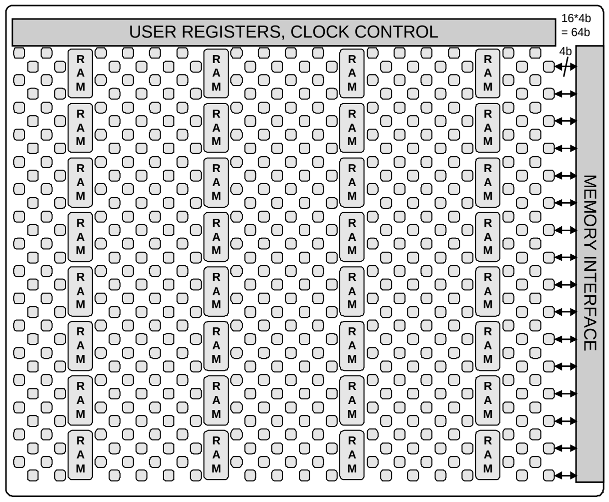

Figure 2 A 512 ALU CHESS array with embedded RAMs

At run time, any switchbox in a CHESS array can be used as a 16W*4b memory. In this mode all of the switches in the switchbox are disconnected, although buses running over the switchbox can still be used. However, if large numbers of switchboxes are used as RAMs the routing capability of the array will be reduced.

To avoid this problem, and to provide higher RAM density than the switchbox memories, the array design also supports embedded block RAMs. These memories are distributed through the array as shown in Figure 2, are attached to user plane wiring buses, and typically are controlled by surrounding ALUs which generate the address and R/W control. In the design shown above the block RAMs are 256W*8b, single ported internally but with separate read and write ports on the interface. Each block RAM takes about the same area as 4 ALUs and switchboxes. The design above provides one block RAM per 16 ALUs, using about 25% more area than the basic array without block RAMs. The block RAMs are appropriate as translation tables and data buffers in media processing algorithms.

Data will be moved on and off the array by connecting to buses on the array edge. Connecting one 4-bit bus on alternate rows of ALUs gives a 64-bit interface on an edge of a 512 ALU array, as shown in Figure 2.

CHESS provides the option of feeding the instruction to an ALU from the output of another ALU, instead of setting the instruction from configuration data. This allows instructions to be changed on a cycle-by-cycle basis, to support predicated execution, to execute sequential 'engines', to implement specialised processors within CHESS, or to give the effect of fine-grain reconfiguration.

We chose a design centre of 512 ALUs in c. 30mm^2 of 0.35µm 4LM CMOS to support our applications. This size was expected to be able to implement the core of a JPEG decoding pipeline.

5.1 ALU logical design

The simplest view of CHESS is as an array of ALUs, and switchboxes that connect them. Each ALU has two 4-bit inputs

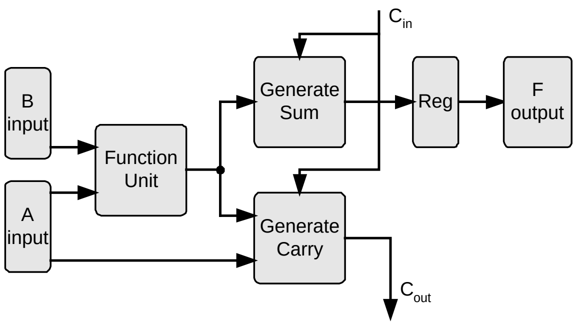

Figure 3 Logical structure of an ALU bitslice

(A and B), and a single bit carry input (C_{in}). The ALU can perform addition, subtraction or logical operations, generating a 4-bit output (F) and a single bit carry output (C_{out}). The instruction is normally held in configuration memory provided for each ALU. Figure 3 shows the logical structure of each bitslice in the ALU.

Table 1 shows the ALU instruction set. All logic operations between A and B are bitwise. Both F and C_{out} can be registered on the ALU output. A variety of carry conditioning options are available.

The carry inputs/outputs carry either data signals (when performing arithmetic for example) or control signals (when performing tests or comparisons on incoming data). For example, the EQUAL instruction tests to see if A==B and drives the result onto C_{out}. The C_{in} input of the ALU can be used to gang multiple EQUAL instructions together when comparing more than 4 bits.

TEST_BITS_AT_0 $C_{out} = C_{in}$ iff AorB=1111 else $C_{out} = 1$	ADD F, $C_{out} = A + B + C_{in}$
TEST_BITS_AT_1 $C_{out} = C_{in}$ iff AandB=0000 else $C_{out} = 1$	SUB F, $C_{out} = A - B - C_{in}$
EQUAL $C_{out} = C_{in}$ iff A == B else $C_{out} = 1$	XOR F = A xor B
INVMUX F = notA / notB (according to C_{in})	XNOR F = A xnor B
MUX F = A/B (according to C_{in})	NOR F = A nor B
AND F = A and B	OR F = A or B
A_AND_NOT_B F = A and notB	B_AND_NOT_A F = B and notA
A_OR_NOT_B F = A or notB	B_OR_NOT_A F = B or notA

Table 1 CHESS ALU instruction set

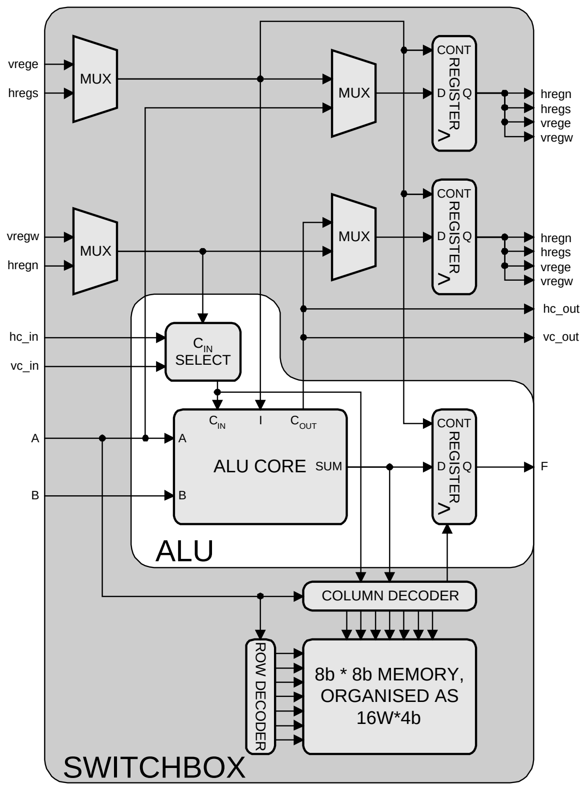

Figure 4 Logical structure of an ALU and switchbox

The TEST_BITS_AT_0/1 instructions can be used to extract a given bit from one operand, by setting the other operand to a suitable mask value. More generally, they test if any of the bits specified in the mask are 0/1. ALUs performing these instructions can also be chained using the carry connections to perform wider tests. The C_{in} input at the LS end should be set to 0. If the test fails in any ALU its C_{out} is driven to 1, which is then propagated through the carry chain to the C_{out} output of the last ALU.

Carry outputs can be used to control ALUs used as multiplexers, or to drive local resets and clock enables. For example, the result of an EQUAL instruction can directly control whether a register is clocked, or reset. Carry signals can be routed through the general routing fabric but also have dedicated, high-speed, local routing paths to their north and east neighbours.

5.2 Switchbox

In fact, CHESS is more complex than described above. As shown in Figure 4, a switchbox contains two 4-bit registers in addition to the wiring switches. These registers are provided to support heavily pipelined designs, needed to maintain clock speed given the delays inherent in reconfigurable wiring. The registers can be clocked always, never, or according to the value of a control signal used as a clock enable. Alternatively, the control signal can be used to set or reset the register. The register can also be made transparent, to be used as a buffer for the long buses. Registers that are never clocked are used to hold constants needed in a computation.

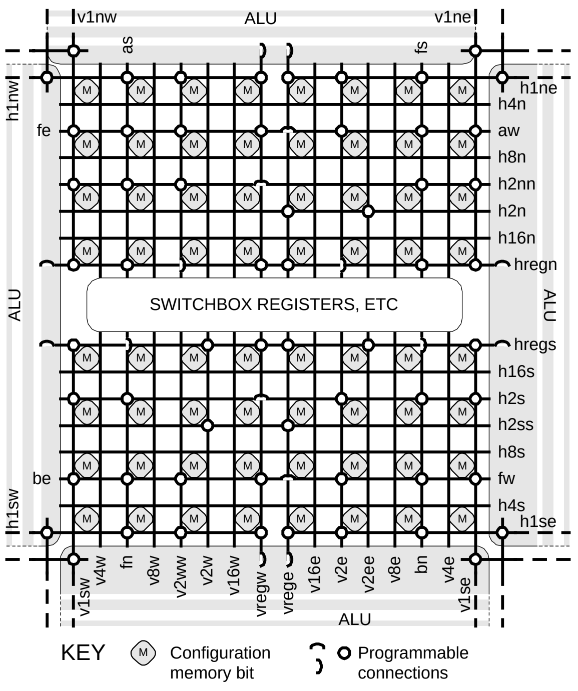

Figure 5 Wiring within a switchbox

The inputs to the registers are each selected among two buses that cross the switchbox, and an additional signal as shown in Figure 4. Selection is done in two stages, which allows the instruction input to the ALU to be derived from an intermediate stage.

The I input to the ALU is an optional instruction input, which shares input connections with the register control input. The ALU instruction is mostly encoded into just 4 bits, so that the instruction can be generated elsewhere and fed into the I input over the normal 4-bit wiring network. By driving the I input to the appropriate ALU op-code, the ALU instruction can be changed on a cycle by cycle basis. This supports many forms of data-dependent execution.

Some aspects of the ALU operation, such as the conditioning of the carry input, are determined by static configuration bits and cannot be dynamically reconfigured.

5.3 The ALU as a memory interface

Instead of being used for computation, each ALU can be combined with an adjacent switchbox to provide a 16W*4b memory (see Figure 4). In this mode the memory address is fed into the ALU A input, write data is fed into the B input, and read data is taken from the F output. C_{in} controls W/notR. The ALU and its output register act as both memory column drivers and sense amplifiers, which reduces the area overhead of providing such small memories.

During reconfiguration these memory blocks and other configuration bits become the bottom levels of the configuration memory, with the second level address decoding (not shown) being done against a hard-wired value related to the ALU location. This second level address decoder is also accessible from user plane wiring, allowing multiple switchbox RAMs to be combined into a larger logical RAM. Yet more ties onto the user plane buses (not shown) are used to tie the memory blocks together into a single memory during configuration.

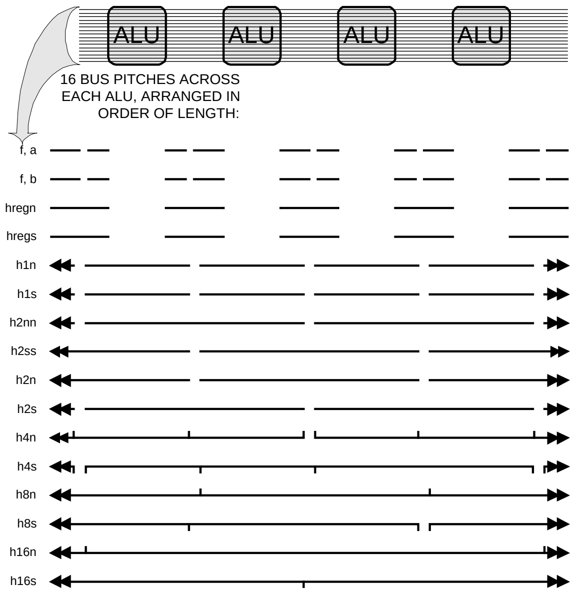

Figure 6 Use of bus pitches across a row of ALUs

5.4 Routing Structure

CHESS has a segmented routing structure, using a range of bus lengths. The routing scheme uses only 50% of the array area, much less than in most FPGAs. We believe that the chessboard style layout is a major factor in achieving good routability in spite of the relatively low proportion of the area used for routing. The following issues drove the routing scheme, shown in Figures 5 and 6:

- With a chessboard style layout the switchbox must be about the same size as the ALU. Thus balances in the IC technology will limit the number of buses supportable. In our case the switchbox holds an 8*8 array of configurable connections and 16 4-bit buses both horizontally and vertically. Each connection contains a memory bit, which programs one connection among the two horizontal and two vertical buses that pass by. Heavily and lightly connected buses are interleaved to reduce contention for these connection points. The connection pattern makes it simple for the lightly connected long buses to be moved to higher layers of metal as these become available in future processes, giving plenty of flexibility in reaching a balanced layout.

- The distribution of lengths of buses is important to the routability and hence the applicability of the array. We have both local and 'long' distributions of buses. Figure 6 shows how the 16 bus pitches across each ALU are used, showing the number and alignment of the buses in increasing order of length.

- Powerful local connectivity is vital to allow most connections in a regular application to be made cheaply. Two bus pitches are used to give each ALU an input and an output port on each of its four sides, providing 8-way nearest-neighbour connectivity via single switches as shown by the grey arrows in Figure 1.

- Two more bus pitches are used to provide centrally placed 'feeder' buses (hregn, hregs, vregw and vrege in Figure 5) for the switchbox register/buffers, which then connect to the long array buses via switches in the sides of the ALUs. Thus switchbox registers can be used for pipelining or buffering long nets to maximise system clock speed. 'Feeder' buses not used for register/buffer access can be used for local connections.

- Two bus pitches are used for L1 buses (crossing one ALU and half a switchbox on either side) which run between feeder buses in adjacent switchboxes. These allow registers to be pipelined cheaply. These L1 buses are connected in the corners of the ALUs, providing linkage between diagonally adjacent switchboxes.

- L2 and longer buses are laid out in staggered pairs (h2n and h2s in Figure 6 for example), so that one bus in the pair is centred on the endpoints of the other bus. This avoids having regions of weak connectivity that would result if all bus ends were aligned.

- Long connections could limit the operating speed of the whole device. To control this, 'long' buses of length 4 and higher powers of 2 support long connections with few intermediate switches. Buses of length 2^n cross 2^n ALUs, and half a switchbox at each end. Each of these has only 5 equally spaced switches to limit capacitive loading. Long buses connect only to the ends of the feeder buses.

- The number of long buses needed per row or column is inversely related to bus length[1]. Over each switchbox (both horizontally and vertically) we have dedicated 4 pitches to L2 buses, and 2 pitches to each of L4, L8 and L16 buses. This is more L8 and L16 buses than needed for a 512 ALU array, but gives headroom for the architecture to scale to larger arrays.

5.5 Physical design

The ALU bitslice uses N-channel pass transistors extensively. The delay through an ALU is around 2.5ns in a 0.35µ 4LM process, equalling the delay through four cascaded L4 buses.

The wiring network uses only N-channel pass transistors for density, and so all logic inputs need level-restoring input buffers.

Metal 2 and 3 are densely packed with vertical and horizontal wiring, with all buses being laid out as simple straight lines for high density. Metal 1 is used for local connections, and metal 4 is used for VDD and GND.

CHESS has been designed to be appropriate for future process generations. The embedded RAMs will become increasingly important for many applications as transistor counts outstrip off-chip bandwidth. A generous supply of long buses, and the ability to add more with little impact on the architecture, will support larger CHESS arrays in future. Generous register provision and lightly connected long buses combine to minimise propagation delays across an array.

Although the chessboard layout of CHESS has the apparent disadvantage of constraining the ALU and switchbox to be about the same size, in practice we have found that there are a number of options for meeting this constraint that do not affect the logical architecture of the device. For example, several blocks of circuitry can easily be moved between the ALU and switchbox to help balance the layout. The long buses, which connect only to the ends of the register buses, could easily be moved to higher layers of metal if these were available.

6. USING CHESS

6.1 Building Datapaths

The ALUs in CHESS were designed to support datapaths in either word-parallel or digit-serial style. In word-parallel style, enough ALUs are used for each operation to cover the full width of the datapath. Where used, the carries are connected in a ripple carry chain. We expect carry propagation to limit clock speed for datapaths wider than 16 bits.

In digit-serial style [9], each operand appears on a digit-wide bus, a digit per clock cycle. In CHESS we will usually choose a digit to be a 4-bit nibble. For most arithmetic operations the operands should appear with the least significant digit first, to allow carries to be propagated to the more significant digits. CHESS provides a switchbox register that can register the carry bit and feed it back to the ALU as needed for nibble-serial operation. The overhead of arranging the data flow in this way can be significant, but once this has been done nibble-serial operation has two major benefits for us:

- Only one ALU is needed to build e.g. an adder for any width of data[2]. Therefore the nibble-serial style can accommodate more complex designs in a given size of array than the word-parallel style.

- Carries pass through only one ALU per clock cycle, and so are fast.

As an example of the compact designs that can be created in the nibble-serial style, Figure 7 shows the layout of a nibble-serial implementation of an 8 point 1D IDCT as used in JPEG decoding [13], on less than ¼ of a 512 ALU CHESS array. The design includes 5 digit-serial multipliers, each multiplying by a 12-bit constant. For 12-bit data this circuit computes an 8 point 1D IDCT every 8 clock cycles[3]. This layout was automatically placed and routed, and does not need any buses longer than L2.

[1] This empirical observation is known as Rent's rule.

[2] The one ALU is used for multiple cycles for data wider than 4 bits of course, with carries being fed from the output back into the input after each nibble addition.

[3] 12-bit data multiplied by 12-bit constants gives 24 bits, needing 6 cycles. We use 8 to ease data rearrangement.

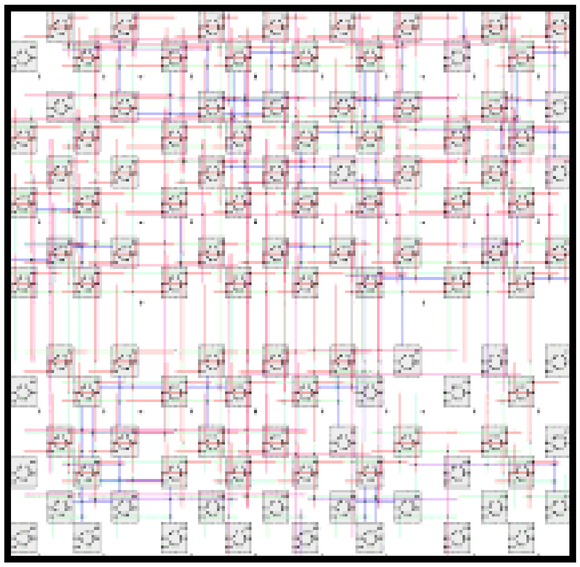

Figure 7 Layout of nibble serial DCT on CHESS

6.2 Control FSMs

Whilst CHESS is primarily optimised for data paths, control circuits also need on-chip support to provide low control latency. Three main techniques can be used:

- Build state transition and output logic from the 4-input, 4-output switchbox RAMs or the 8-input, 8-output embedded block RAMs. This technique is useful for small FSMs, but requires the logic to be partitioned between multiple communicating RAMs to avoid exponential growth with the number of states.

- Implement FSMs as adders with feedback loops, giving counters by default. Add multiplexers in the feedback path to allow conditional branching to states. This technique is useful for cyclic state machines with few inputs, often found in digit-serial designs. However, it scales badly as the number of control inputs increases.

- Small CPUs can be constructed if appropriate, using sequence tables of instructions feeding functional units. Such CPUs can implement very complex operations, but will often have a response latency of many clock cycles.

The key to implementing efficient state machines in CHESS will be to distribute small controllers to the places they are needed, rather than building large centralised controllers.

6.3 Using on-chip memory

Applications can often be restructured to reduce external memory traffic by using CHESS' on-chip RAM. For example, applying a 5*5 convolution to a large image scanned in raster order requires 4 lines of intermediate results to be held before a given block of the convolution can be completed. For large images memory traffic of these intermediate values to and from external DRAM is likely to limit system performance. However, techniques of problem subdivision (e.g. convolving in 256*256 pixel sub pictures and recombining) can be effective if moderate line-length buffers are available on chip. CHESS' block RAMs are suitable for such sizes.

7. PREVIOUS WORK

As we report on performance, we contrast with previous work on reconfigurable architectures for computation. These can be categorised in two broad camps:

- Architectures that provide traditional FPGA-style routing, allowing connections between arbitrary locations in a 2D array. Within this constraint, the functional blocks and routing structure are optimised for datapath applications.

- Architectures designed to support general-purpose computing models with *virtual hardware*. This involves paging circuit modules in and out of a limited physical device as needed, preferably while the rest of the device is still computing. Efficient hardware virtualisation requires run-time reconfiguration (RTR) to avoid repeated stalling of the application, and also requires that the configuration data be small so that it can be changed quickly. Architectures intended to support RTR often use processors or specialised function units to reduce the amount of configuration data used to specify the function, and a linear organisation to reduce the amount of configuration data used for routing.

7.1 Architectures with FPGA-style routing

Like CHESS, DP-FPGA [5, 6] shared configuration bits between 4 bit-slices, reducing the configuration overhead for both wiring switches and the function blocks. A dedicated carry chain within each 4-bit logic block supported arithmetic applications. In other respects DP-FPGA differed from CHESS:

- DP-FPGA used LUTs to implement function blocks. These are more general than the ALUs used in CHESS, but need more configuration bits to control them. Because of this it would be hard to provide dynamic instruction inputs to a DP-FPGA logic block in the way that CHESS does.

- The routing structure of DP-FPGA was not described in full, but was biased to support horizontal data and separate vertical control connections. In contrast, the wiring structure of CHESS has been made as uniform as possible, with equal densities of horizontal and vertical buses. This is intended to support the routing of datapaths that are much more complex than simple pipelines.

- DP-FPGA provided separate routing structures for data and control. In contrast, CHESS shares one routing structure between data and carry/control connections[4]. This means that CHESS uses only 1 wire in a 4-bit bus used for a carry/control connection, but we prefer that to incurring the overhead of a separate control network.

- DP-FPGA included dedicated shift blocks to compensate for the 4-bit granularity of the routing structure. CHESS has no direct equivalent, but relies on ALU instructions and 4-input, 4-output switchbox RAMs to implement shifts where needed.

[4] Excepting fast carry paths, which are only used for carry/control.

7.2 Architectures to support virtual hardware

The major issue in supporting virtual hardware efficiently is how to provide a stream of configuration data to specify the changes in function at run time.

Several architectures source configuration data from external memory. PipeRench [4] uses incremental RTR to make the best use of external memory bandwidth, and is based on homogeneous stripes of ALUs with only global and nearest-neighbour interconnect. The restricted interconnect limits the amount of configuration data needed to specify the connection pattern.

Colt [3] is a 2D coarse-grained array, including a system of wormhole RTR that provides linear RTR within a 2D array. As in PipeRench, incremental reconfiguration is used to make the best use of the external memory bandwidth.

Other architectures address the constraints on reconfiguration bandwidth by providing multiple local configurations in local RAMs, and distributing control signals to activate the appropriate local configurations during the computation. However, the cost of storing the local configurations still constrains these architectures to use minimal amounts of configuration data. RaPiD [8] and REMARC [11] are examples of this style of architecture.

RaPiD is a linear array of arithmetic-oriented units, including ALUs, multipliers, registers and RAMs. The restriction to linear pipelines allows RaPiD to distribute the configuration control in a control pipeline in parallel with the data pipeline.

REMARC is a 2D array of processors, each with a local nano-instruction RAM. The nano instructions are addressed by a global program counter, and control the processor operation and the routing of data, giving an effect similar to RTR in an FPGA. The 'reconfiguration' (changing the value of the global program counter) takes place across the whole array at once, but is limited to the current contents of the nano instruction RAMs. The routing on REMARC is more limited than on an FPGA, being limited to nearest neighbour connections, and global row and column buses.

All of these architectures have much less flexible routing structures than general purpose FPGAs, so that the configuration data can be kept small enough to allow hardware virtualisation. MATRIX [7] introduced to reconfigurable computing the powerful concept of generating configuration data from *within* the array, and so side-steps the constraints of limited configuration storage and bandwidth to some extent by allowing configuration data to be sourced internally on a selective basis. As a result, MATRIX is the only architecture in this group that can offer an interconnect structure that approaches that of an FPGA.

Of the architectures in this group, MATRIX is the closest to CHESS. Both architectures are based on a 2D array of coarse grain processors whose instructions can be generated within the array, and both aim to retain most of the routing flexibility of a general purpose FPGA. However, CHESS does not allow the routing to be dynamically reconfigured as MATRIX does. As a result, CHESS needs only a 4-bit instruction input to an ALU, making dynamic instruction inputs much cheaper than in MATRIX.

Although CHESS' routing cannot be dynamically reconfigured, its modest number of configuration bits allows it to be partially or globally reconfigured off-line with modest downtime. Also, CHESS is dense enough to support practical applications in one configuration of a small array, which reduces the need for it to support hardware virtualisation.

8. PERFORMANCE

As we set out with the main goal of achieving high computational density in the design, let us quantify the computational density achieved by CHESS, and compare it with other FPGAs. The area of the basic cell (ALU plus switchbox) is 0.045 mm^2 in a process with 0.35µ feature size (D). Simulation results predict an operating speed of 200MHz for *tight pipelines* (one ALU plus 4 buses, or two ALUs with direct connections, between two clocked registers).

For the purpose of comparing these figures with existing FPGAs (regardless of feature size), we use the techniques developed in [7]. Area measures are normalised by expressing them in units of λ^2, where λ is half of the feature size. This gives the relative areas as if all chips were built in the same process.

Normalising the speed is more complex. The delay of a minimally loaded inverter has scaled almost as the inverse of feature size for processes from 1.2µ to 0.35µ. [10]. However, delays due to interconnect have not been reducing in proportion to feature size due to increasing RC delays [1], and this effect is significant for FPGAs. This makes it hard to quantify the clock speeds that might be expected for the other FPGAs if they had been built in a 0.35µ process.

For this reason we show comparative performance figures in terms of operations per Mλ^2 per cycle, implicitly assuming that all architectures could be clocked at the same speed if built in the same process. We believe that in practice typical designs in CHESS could be clocked faster than their equivalents in the other architectures, and that in reality CHESS has an additional performance advantage beyond that shown in the table. Using data from [7] as a basis for the size and speed of other FPGAs, we find the following normalised areas, where the figures for CHESS refer to a pure array of ALUs and switchboxes with no embedded RAMs:

	X3K (CLB)	X4K (CLB)	X6K (CELL)	CHESS (ALU + Sbox)
Process (µ)	1.2	1.2	0.6	0.35
λ (µ)	0.6	0.6	0.3	0.18
Area (mm^2)	0.47	0.45	0.020	0.043
Area/M λ^2	1.30	1.25	0,22	1.41
ops/cell/cycle:				
bit ops	2	32	1	68
lut4 ops	2	2	0.17	4
alu ops	1	2	0.33	4
mul ops	0.67	0.67	0.25	4
ops/M λ^2/cycle:				
bit ops	1.5	25	4.6	48
lut4 ops	1.5	1.6	0.8	2.8
alu ops	0.8	1.6	1.5	2.8
mul ops	0.5	0.5	1.1	2.8

In this chart, four measures of computational density are derived from the cell area:

1. bit ops, the number of storage bits in the cell.
2. lut4 ops, the number of 4-input, one output LUT operations that the cell can perform.
3. alu ops, the number of one bit additions that the cell can perform.
4. mul ops, the number of 1-bit x 1-bit multiply operations that the cell can perform.

CHESS has a substantial performance advantage over all of the other FPGAs quoted on all of these measures, even excluding the benefits of the higher clock speeds that we believe will be made possible by using CHESS' switchbox registers to pipeline long connections. We plan to characterise the clock speeds that CHESS can achieve in more detail by mapping realistic applications to the architecture.

9. CONCLUSIONS

CHESS achieves high computational density primarily through the use of 4-bit buses for routing, and the adoption of a chessboard style layout. Together, these allow the area devoted to configuration bits and routing switches to be about 50% of the area of a basic CHESS array, leaving the rest available for user-visible functional units.

Raw performance is only useful if the applications of interest can be mapped to the array efficiently. CHESS' flexibility in application mapping is largely due to the ability to feed ALUs with instruction streams generated within the array, generous provision of embedded block RAMs, and the ability to trade routing switches for small memories.

By focussing on requirements for high arithmetic performance at low cost in media processing applications, CHESS makes trade-offs quite different from those applicable to general purpose FPGAs. The result is a new class of device, the Reconfigurable Arithmetic Array (RAA), with higher computational density than FPGAs on several measures, and some unusual forms of flexibility.

10. ACKNOWLEDGEMENTS

The authors would like to thank Charlie Tan and Fred Perner of HP's ULSI Lab in Palo Alto for their numerous valuable contributions towards this work. We would also like to thank John Lumley and Nick Wainwright of HP Laboratories, Bristol for their extensive support throughout the project.

11. REFERENCES

[1] HB Bakoglu, "Circuits, Interconnections, and Packaging for VLSI", Pub. Addison Wesley, 1990. ISBN 0-201-06008-6.

[2] P Bertin, D Roncin and J Vuillemin, "Programmable Active Memories: a Performance Assessment", DEC PRL Report 24. 1993. Available from: http://www.research.digital.com/PRL/publications/pam.html

[3] R Bittner, P Athanas, M Musgrove, "Colt – An Experiment in Wormhole Run-Time Reconfiguration", Proc SPIE Photonics East 96.

[4] S Cadambi, J Weener, S Goldstein, H Schmit and D Thomas, "Managing Pipeline-Reconfigurable FPGAs", Proc. ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays (FPGA98), Feb 1998.

[5] Don Cherepacha and David Lewis, "A Datapath Oriented Architecture for FPGAs", Proc. ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays (FPGA94), Feb 1994.

[6] Don Cherepacha and David Lewis, "DP-FPGA: An FPGA Architecture Optimized for Datapaths", VLSI Design V4-4, 1996, pp 329-343

[7] A DeHon, "Reconfigurable Architectures for General-Purpose Computing", MIT, Artificial Intelligence Laboratory, AI Technical Report No. 1586, 1996.

[8] C Ebeling, D Cronquist and P Franklin, "RaPiD – Reconfigurable Pipelined Datapath", Proc. 6th Int'l Workshop on Field-Programmable Logic and Applications, FPL96. pp. 126-135. Pub Springer

[9] RI Hartley and KK Parhi, "Digit Serial Computation", Pub: Kluwer, 1995.

[10]C Mead, "Scaling of MOS Technology to Submicrometre Feature Sizes", Journal of VLSI Signal Processing, V 8, N 1, pp. 9-26, 1994.

[11]T Miyamori and K Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications", IEEE Symp. on FPGAs for Custom Computing Machines (FCCM98), Apr 1998.

[12]Orca Series 3 datasheets are available from: http://www.lucent.com/micro/fpga/

[13]WB Pennebaker and JL Mitchell, "JPEG Still Image Data Compression Standard", Pub: Van Nostrand, 1992. See p52 - flowgraph for DCT adapted from Arai, Agui & Nakajima.

[14]Vantis VF1 family datasheets are available from: http://www.vantis.com/

[15]J. Vuillemin, P. Bertin , D. Roncin, M. Shand, H. Touati and P. Boucard, "Programmable Active Memories: the Coming of Age", IEEE Trans. on VLSI, Vol. 4, NO. 1, pp. 56-69, 1996.

[16]Xilinx 4000 series datasheets are available from: http://www.xilinx.com/products/products.htm